Publication Number 29-747

MODEL 3210 PROCESSOR

USER'S MANUAL

PERKIN-ELMER

Computer Systems Division
2 Crescent Place
Oceanport, N.J. 07757

The Perkin-Elmer Corporation, Computer Systems Division 2 Crescent Place, Oceanport, New Jersey 07757

TABLE OF CONTENTS

CHAPTERS (Continued)

CHAPTERS (Continued)

CHAPTERS (Continued)

CHAPTERS (Continued)

CHAPTERS (Continued)

APPENDIXES

FIGURES

FIGURES (Continued)

TABLES

PREFACE

The Model 3210 Processor User's Manual provides programming and operating information for the System. The programmer is provided with information on the 32-bit system architecture and the unique memory management scheme, as well as a description of each instruction in the repertoire. The instruction descriptions include valuable system-related information presented in the form of programming notes and instruction examples.

Information pertaining to the system control terminal is given to facilitate program preparation and execution for the system programmer and operator.

CHAPTER 1
SYSTEM DESCRIPTION

1.1 INTRODUCTION

The Model 3210 processor is designed to meet the needs for high performance and reliability in a 32-bit minicomputer. The architecture has improved error recovery capabilities for those applications where fault tolerance is a necessity, and allows direct addressing of up to 4 megabytes of memory implemented in MOS with Error-Correction Code (ECC).

Through the use of 32-bit general registers and a comprehensive instruction set, the Model 3210 processor provides fullword data processing power and direct memory addressing up to a limit of 8 megabytes. The system is shown in block diagram form in Figure 1-1. The instruction set includes:

- halfword and fullword arithmetic and logical operations
- optional single-precision and double-precision floating-point
- list processing
- cyclic redundancy checking
- bit and byte manipulations
- alphanumeric and decimal character string processing
- decimal/binary conversions
- instructions designed to improve operating system performance

With this enriched repertoire and direct memory addressing, coding and debugging time is reduced to a minimum.

Eight sets of 16 32-bit general registers are provided. Register set selection is controlled by bits in the Program Status Word (PSW). Register-to-register instructions permit operations between any of the 16 registers in the current set, eliminating redundant loads and stores. The multiple register set organization eliminates the overhead incurred in saving and restoring registers when responding to interrupts.

The Memory Address Translator (MAT) provides automatic program segmentation, relocation, and protection. The protect mode enables detection of privileged instructions. These two features are invaluable in process control, data communication, and time-sharing operations because they prevent a running program from interfering with the system integrity.

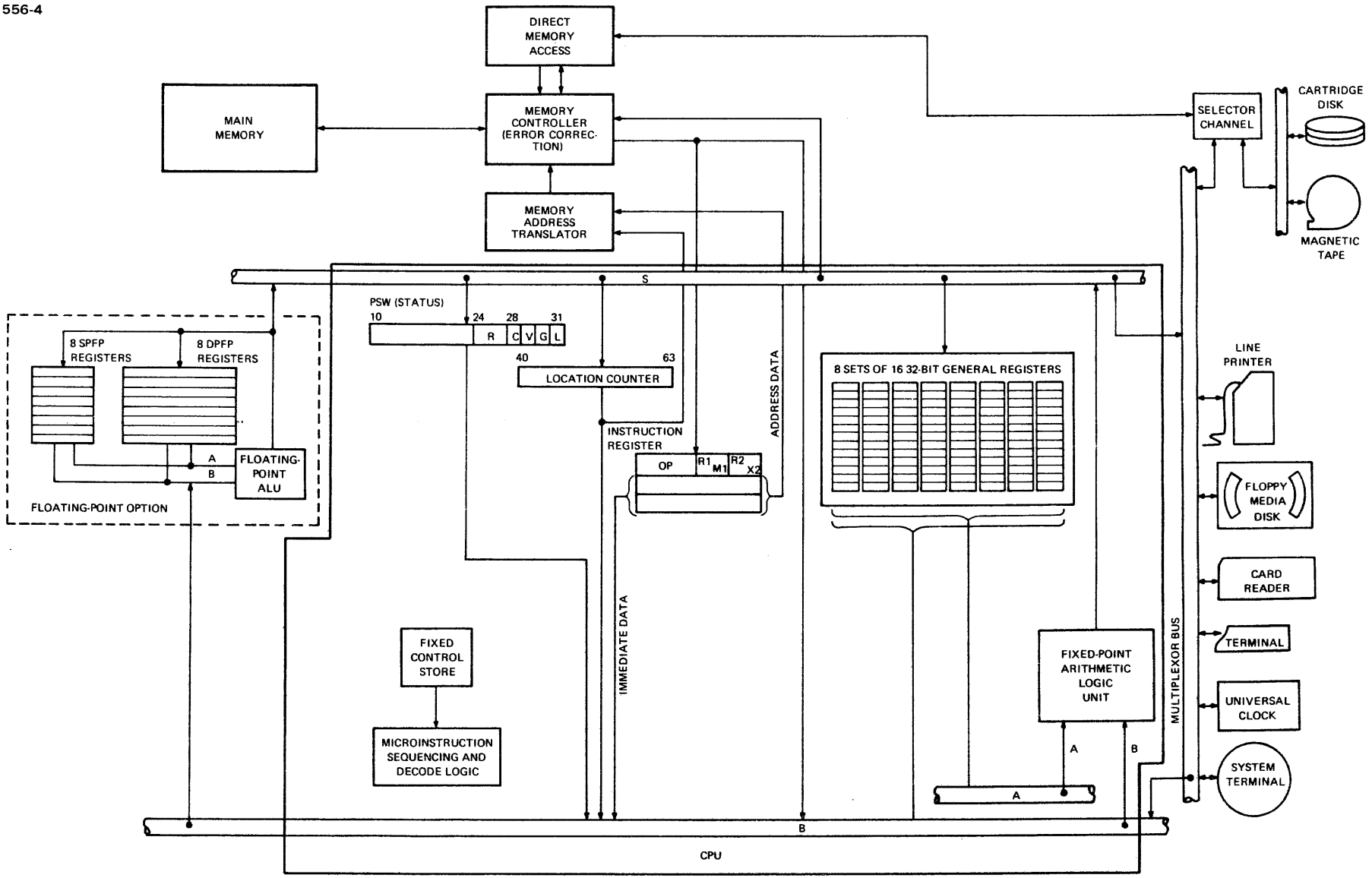


Figure 1-1 Model 3210 Processor Block Diagram

The Model 3210 supports up to 4 Mb of directly addressable MOS memory. Error correction is standard and is performed across every 32-bit fullword in memory using a 7-bit modified Error-Correcting Code (ECC). All single-bit errors are detected and corrected; all double-bit errors and most multiple-bit errors are detected. The memory error logger identifies the memory module reporting a fault and indicates the location of the faulty memory chip.

In addition to conventional means of programmed I/O, the processor automatically acknowledges all I/O interrupts and performs much of the required overhead before activating an interrupt service routine. The auto driver channel can perform data transfers with character translation, longitudinal or cyclic redundancy checking, and data buffer chaining without interrupting the running program.

Refer to the following manuals for further information:

Common Assembly Language (CAL) User's Manual, Publication Number 29-640

ESELCH Programming Manual, Publication Number 29-529

EDMA Bus Universal Interface Instruction Manual, Publication Number 29-423

Model 3210 Maintenance Manual, Publication Number 47-022

1.2 PROCESSOR

The Central Processing Unit (CPU), or processor, controls activities in the system. (See Figure 1-1.) It executes instructions in a specific sequence and performs arithmetic and logical functions. Included in the processor's components are the following:

- Program status word register
- General registers
- Hardware multiply and divide

1.2.1 Program Status Word (PSW)

The 64-bit Program Status Word (PSW) defines the state of the processor at any given time. (See Figure 1-2.)

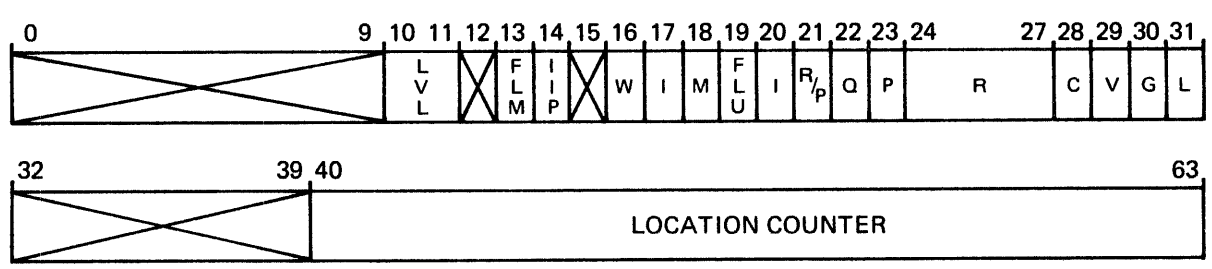


Figure 1-2 Program Status Word

Bits 0:31 are reserved for status information and interrupt masks. Bits 32:63 contain the location counter. Unassigned PSW bits must not be used and must always be zero. Status information and interrupt mask bits are defined as follows:

Bits 0:9	Reserved	Must be zero
Bits 10:11	LVL	Memory access level
Bit 12	Reserved	Must be zero
Bit 13	FLM	Floating-point arithmetic masked mode
Bit 14	IIP	Interruptible instruction in progress
Bit 15	Reserved	Must be zero
Bit 16	W	Wait state
Bit 17	I	I/O interrupt mask
Bit 18	M	Machine malfunction interrupt mask
Bit 19	FLU	Floating-point arithmetic underflow mask
Bit 20	I	I/O interrupt mask
Bit 21	R/P	Relocation/protection interrupt mask
Bit 22	Q	System queue interrupt mask
Bit 23	P	Protect mode
Bits 24:27	R	Register set select bits
Bits 28:31	C,V,G,L	Condition code
Bits 32:39	Reserved	Must be zero
Bits 40:63		Program address (location counter)

Refer to Chapter 10 for details on the interrupt mask bits.

1.2.1.1 Register Set Select (R)

Bits 24:27 of the PSW are used to designate the current register set. Register sets are numbered 0 through 15. The processor has eight sets of general registers. (See Figure 1-3.)

558

REGISTER SET NUMBER	DESIGNATION
0 1 2 3	RESERVED FOR INTERRUPTS
4 5 6	MAY BE ALLOCATED BY THE OS FOR GENERAL PURPOSE USE.
7 8 9 10 11 12 13 14	UNIMPLEMENTED SETS
15	GENERAL PURPOSE

Figure 1-3 Register Set Numbering

1.2.1.2 Condition Code (C,V,G,L)

Bits 28:31 of the PSW contain the condition code. As part of the execution of certain instructions, the state of the condition code may be changed to indicate the nature of the result. Not all instructions affect the condition code. The state of the condition code may be tested with conditional branch instructions. Each bit in the condition code is set if the corresponding condition occurred as a result of the last instruction that affected the condition code. The normal interpretation of these bits is:

C	V	G	L	
1	0	0	0	Arithmetic carry, borrow, or shifted carry
0	1	0	0	Arithmetic overflow
0	0	1	0	Greater than zero
0	0	0	1	Less than zero

1.2.1.3 Location Counter

The location counter contains the address of the instruction currently being executed by the processor, and points to that instruction until it has successfully completed execution. Once this execution is completed, the location counter is incremented by 2, 4, 6, 8, 10, or 12 (depending upon the instruction executed), and the next instruction is fetched. In the case of a branch instruction, the location counter is loaded with the address to which control is being transferred, and the next instruction is fetched from that address.

If an instruction is not successfully completed due to a fault or other interrupting condition, the location counter contains the address of the faulting or interrupted instruction. When a program interruption is due to an incorrect branch address, the location counter contains the branch address and not the location of the branch instruction.

1.2.2 General Registers

The processor has eight register sets, numbered 0 through 6, and 15 (see Figure 1-3). There are 16 registers in each set and each register is 32 bits wide. Register set selection is determined by the state of bits 24:27 of the current PSW. Registers 1 through 15 of any set may be used as index registers.

When an interrupt occurs, the processor loads pertinent information into preselected registers of the register set selected by the new PSW. For details of this operation, refer to Chapter 10.

1.2.3 Floating-Point Registers

There are eight optional single-precision floating-point registers, each 32 bits wide. These registers are identified by the even numbers 0 through 14.

There are eight optional double-precision floating-point registers, each 64 bits wide. These registers are also identified by the even numbers 0 through 14 and are separate from the single-precision floating-point registers. Floating-point operations must always specify the registers with even numbers.

1.3 PROCESSOR INTERRUPTS

The PSW that is loaded in the processor at any point in time is called the current PSW. If either the status word or both the location counter and status word are changed, a status switch is said to have occurred. This status switch can be caused explicitly by executing special instructions, or it can be forced to occur by an interrupt or fault. At the time of a status switch, the current PSW that is saved is called the old PSW. The PSW that replaces the current PSW is called the new PSW.

Interrupt conditions cause the entire PSW to be replaced by a new PSW, thus breaking the usual sequential flow of instruction execution. When an interrupt condition occurs, the processor saves its current PSW either in memory or in a pair of general registers belonging to the register set selected by the new PSW. It loads information related to the interrupt condition in other registers of this same set. A new PSW is loaded from a memory location reserved for the specific interrupt condition. The immediate interrupt is an exception to the rule. In this case, the status portion of the new PSW, bits 0:31, is forced to a preset value, and the location counter is loaded from a memory location reserved for that interrupting device. Refer to Chapter 10 for details of interrupt processing.

1.4 RESERVED MEMORY LOCATIONS

Physical memory locations X'0'-X'2CF' are called reserved memory locations. These locations contain the various new PSWs and other information needed to handle interrupts.

X'000000'-X'00001F'	Reserved; must be zero
X'000020'-X'000027'	Machine malfunction interrupt old PSW
X'000028'-X'00002B'	Used by console service microcode
X'00002C'-X'00002F'	Machine malfunction LM block start address
X'000030'-X'000037'	Illegal instruction interrupt new PSW
X'000038'-X'00003F'	Machine malfunction interrupt new PSW
X'000040'-X'000043'	Machine malfunction status word
X'000044'-X'000047'	Machine malfunction virtual (program) address
X'000048'-X'00004F'	Arithmetic fault interrupt new PSW
X'000050'-X'00007F'	Boctstrap loader and device definition table
X'000080'-X'000083'	System queue pointer
X'000084'-X'000087'	Power fail save area pointer
X'000088'-X'00008F'	System queue service interrupt new PSW

X'000090'-X'000097'	Relocation/Protection interrupt new PSW
X'000098'-X'00009B'	Supervisor call new PSW status
X'00009C'-X'0000BB'	Supervisor call new PSW location counter values (16 halfwords)
X'0000BC'-X'0000BF'	Reserved; must be zero
X'0000C0'-X'0000C7'	Reserved; must be zero
X'0000C8'-X'0000CF'	Data format fault new PSW
X'0000D0'-X'0002CF'	Interrupt service pointer table
X'0002D0'-X'0004CF'	Expanded interrupt service pointer table
X'0004D0'-X'0008CF'	Expanded interrupt service pointer table

These reserved locations play an important role in both interrupt and input/output processing. (Refer to Chapters 9 and 10.)

All location counter values are subject to MAT relocation if the new PSW enables the MAT (bit 21=1). All other pointers contain absolute addresses not subject to MAT relocation.

1.5 DATA FORMATS

The processor performs logical and arithmetic operations on single bits, 8-bit bytes, 16-bit halfwords, 32-bit fullwords, and 64-bit doublewords. This data may represent a fixed-point number, a floating-point number, logical information, a bit or byte array, or a decimal or alphanumeric byte string.

1.5.1 Fixed-Point Data

Fixed-point arithmetic operands may be either 16-bit halfwords or 32-bit fullwords. In fullword multiply and divide operations, 64-bit operands are manipulated. Fixed-point data is treated as 15-bit signed integers in the halfword format. Positive numbers are expressed in true binary form with a sign bit of zero. Negative numbers are represented in two's complement form with a sign bit of one. The numerical value of zero is represented with all bits zero. Refer to Chapter 5 for details of fixed-point data representation.

In fixed-point arithmetic and logical operations between a fullword register and a halfword operand, the halfword operand is expanded to a fullword by propagating the most significant bit into the high order bits before the operation is started. This permits the use of halfword to fullword operations with consistent results and provides space economy, since small values do not require fullword locations.

Arithmetic operations on fixed-point halfword quantities may produce results not entirely consistent with those obtained in a 16-bit processor. If this problem exists, the Convert to Halfword Value Register (CHVR) instruction may be used to adjust the result and the condition code, making them consistent with the same operations in a 16-bit processor.

1.5.2 Floating-Point Data

A floating-point number consists of a 7-bit exponent in excess-64 notation and a signed fraction. The quantity expressed by this number is the product of the fraction and the number 16 raised to the power represented by the exponent. Each floating-point value requires a 32-bit fullword or a 64-bit doubleword, of which eight bits are used for the sign and exponent. The remaining bits are used for the fraction. Refer to Chapter 6 for details of floating-point data representation.

Floating-point operations take place between the contents of a floating-point register and another floating-point register, a floating-point operand contained in a fullword or doubleword in memory, or a general register or pair of general registers.

1.5.3 Logical Data

Logical operations manipulate 8-bit bytes, 16-bit halfwords, and 32-bit fullwords. In addition, it is possible to perform logical operations on single bits located in bit arrays. Refer to Chapter 3 for details of logical data representation.

1.5.4 Decimal String Data

Decimal strings are strings of consecutive bytes in memory that begin and end on byte boundaries. Information contained in a decimal string may represent packed or unpacked decimal data. Refer to Chapter 7 for details of decimal data formats and operations.

1.5.5 Alphanumeric String Data

Alphanumeric strings are strings of consecutive bytes in memory that begin and end on byte boundaries. Information contained in an alphanumeric string may represent any character stream including decimal string data. Refer to Chapter 7 for details of alphanumeric string data format and operations.

1.6 DATA ALIGNMENT

The following discussion is unique to the Model 3210 implementation and is presented for information only. Any program that misuses a processor feature by taking advantage of a peculiarity of one implementation may not work on a different implementation.

Locations in main memory are numbered consecutively, beginning at address '000000'. Although memory is addressable and alterable to the byte level, machine accesses to memory involve only halfwords or fullwords. These instructions requiring a single byte access actually access a halfword and then manipulate the appropriate byte within the halfword.

Memory can be accessed only to the halfword level; therefore, bit 31 of the address is truncated at the memory. A halfword fetch at address '000051' and a fetch at address X'000050' produce the same halfword. There is no warning mechanism telling the program that it is fetching halfwords on the odd byte boundary.

The CAL Assembler generates an error flag if it sees halfword operations directed to an odd byte address or if it sees fullword operations directed to other than a fullword address.

Bytes of information are addressed by their specific hexadecimal address. Two bytes form a halfword. Halfwords have an even address, the address of the left-most byte in the pair. Two halfwords comprise a fullword. A fullword address is a multiple of four (4 bytes) and is the address of the left-most halfword in the pair. The hardware actually truncates the least significant two address bits on fullword accesses, forcing proper alignment. A data format fault is generated if a fullword access is directed to an address that has bit 30 or 31 set; or if a halfword store is directed to an address that has bit 31 set.

1.7 INSTRUCTION ALIGNMENT

User-level instructions are always aligned on halfword boundaries. Any halfword address is valid regardless of the length of the instruction word. The CAL assembler generates boundary errors if the assembled location counter for an instruction becomes odd. At the machine level, an attempt to make the instruction location counter odd by branching or causing a status switch is ignored by the hardware. In the Model 3210, location counter bit 31 is not implemented and is therefore always zero. Thus, a branch to address X'000051' causes the location counter to be set to X'000050'.

1.8 INSTRUCTION FORMATS

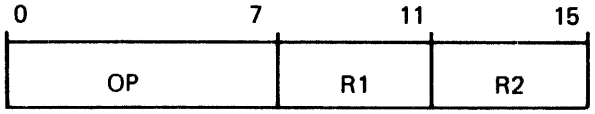
1.8.1 Introduction

Instruction formats provide a concise method of representing required operations for easy interpretation by the processor. Figure 1-4 shows the eight basic formats. The following is a list of abbreviations and their meanings as used in Figure 1-4.

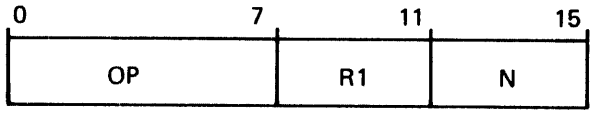
OP	Operation code
R1	First operand register
R2	Second operand register
N	A 4-bit immediate value
X2	Second operand single index register
D2	Second operand displacement
FX2	Second operand first index register
SX2	Second operand second index register
A2	Second operand direct address
I2	Second operand immediate value
L1	Specifies the length of the first operand
L2	Specifies the length of the second operand
CPMOD	Specifies a particular instruction within the class specified by OP
ADD1	The effective first operand address
ADD2	The effective second operand address

Many instructions may be expressed in two or more formats. This feature provides flexibility in data organization and instruction sequencing. When working with the Common Assembly Language (CAL) assembler, it is unnecessary to specify the instruction format. The assembler selects the most economical format and supplies the required bits in the machine code. When double indexing is required, the assembler always chooses the RX3 format. (Refer to the Common Assembly Language (CAL) Manual, Publication Number 29-640.)

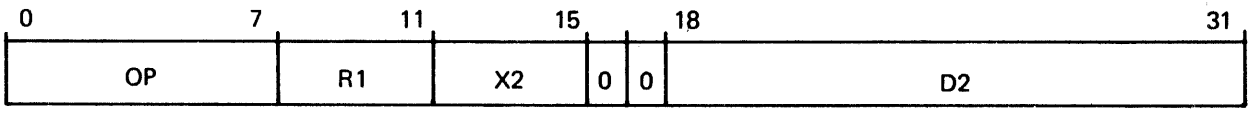
557-I REGISTER-TO-REGISTER (RR)



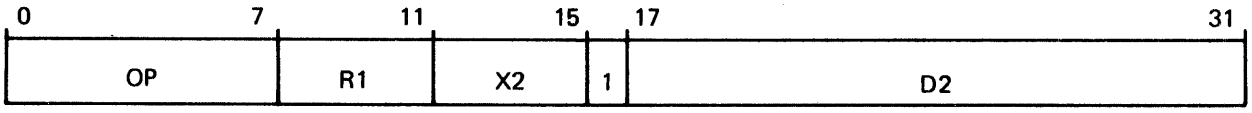
SHORT FORMAT (SF)



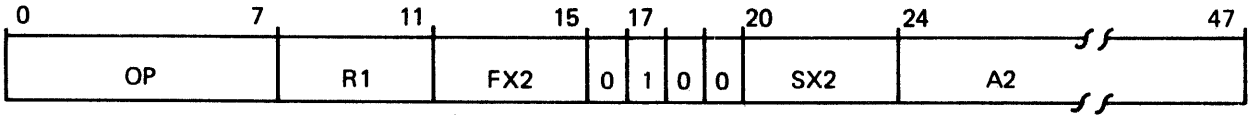
REGISTER AND INDEXED STORAGE (RX1)



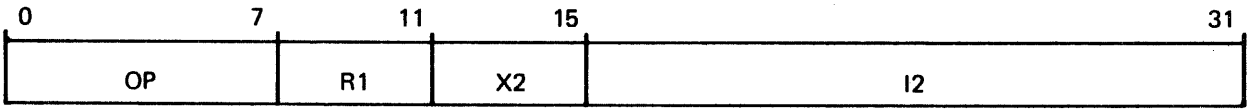
REGISTER AND INDEXED STORAGE 2 (RX2)



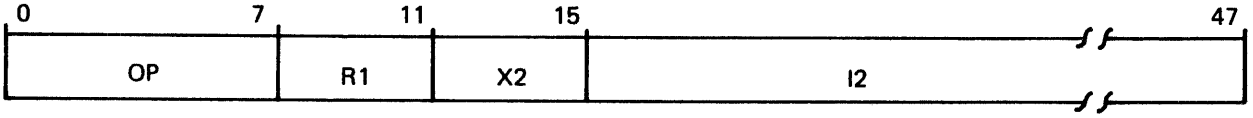
REGISTER AND INDEXED STORAGE 3 (RX3)



REGISTER AND IMMEDIATE STORAGE 1 (RI1)



REGISTER AND IMMEDIATE STORAGE 2 (RI2)



REGISTER AND INDEXED STORAGE, REGISTER AND INDEXED STORAGE (RXRX)

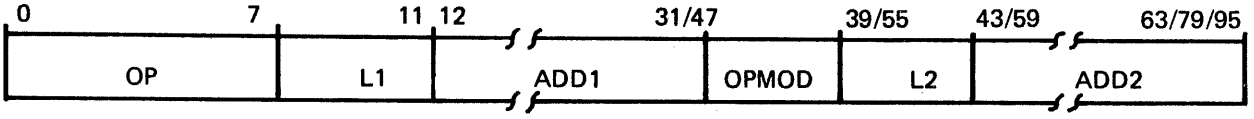


Figure 1-4 Instruction Formats

1.8.2 Branch Instruction Formats

Branch instructions use the RR, SF, and all variations of the RX formats. In the conditional branch instructions, however, the R1 field does not specify a register; instead, it contains a mask value (labeled M1 in the instruction descriptions). This mask value is tested with the condition code. The CAL assembler provides a series of extended branch mnemonics, which makes it possible to specify a conditional branch without specifying the mask value explicitly.

1.8.3 Programming Examples

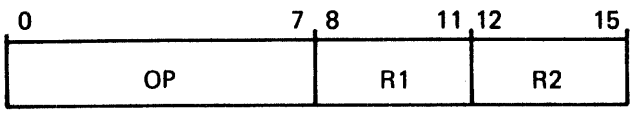
Each of the following examples refers to the sample assembly language program shown in Figure 1-5. Note the use of symbolic equates for general registers. Machine code generated and the result of each instruction are dependent upon the physical and logical placement of the instructions, respectively.

```
560       SERIES 3200 INSTRUCTION FORMAT EXAMPLES                    PAGE    1  18:21:44  02/09/79

    PROG= S3200       ASSEMBLED BY CAL 03-066R05-01   (32-BIT)

                                          1  S3200     PROG  SERIES 3200 INSTRUCTION FORMAT EXAMPLES
                                          2            CROSS
                                          3            NORX3


            0000 0005                     5  R5        EQU   5                 GENERAL REGISTER 5
            0000 0006                     6  R6        EQU   6                 GENERAL REGISTER 6
            0000 0007                     7  R7        EQU   7                 GENERAL REGISTER 7
            0000 0008                     8  R8        EQU   8                 GENERAL REGISTER 8
            0000 0009                     9  R9        EQU   9                 GENERAL REGISTER 9
            0000 000A                    10  R10       EQU   10                GENERAL REGISTER 10
            0000 000B                    11  R11       EQU   11                GENERAL REGISTER 11

000000I     245E                         13  SF        LIS   R5,14             (R5) = *0000000E*

000002I     0865                         15  RR        LR    R6,R5             (R6) = *0000000E*

000004I     4050 1000                    17  RX1.EX1   STH   R5,X*1000*        (X*1000*) = X*000E*

000008I     4056 0FF2                    19  RX1.EX2   STH   R5,X*0FF2*(R6)    (X*1000*) = X*000E*

00000CI     4050 8004 =000014I           21  RX2.EX1   STH   R5,LOC1           (LOC1) = X*000E*
000010I     4300 8004 =000018I           22            B     RI1.EX1
000014I     0000 0000                    23  LOC1      DC    F*0*              TWO HALFWORDS OF STORAGE

000018I     C890 8000                    25  RI1.EX1   LHI   R9,X*8000*        (R9) = Y*FFFF8000*

00001CI     C895 8000                    27  RI1.EX2   LHI   R9,X*8000*(R5)    (R9) = Y*FFFF800F*

000020I     F8A0 0000 8000               29  RI2.EX1   LI    R10,X*8000*       (R10) = Y*00008000*

000026I     F8BA 0001 7FFE               31  RI2.EX2   LI    R11,Y*17FFE*(R10) (R11) = Y*0001FFFE*

00002CI     4050 FFE4 =000014I           33  RX2.EX2   STH   R5,LOC1           (LOC1) = X*000E*

000030I     4056 FFD2 =000006I           35  RX2.EX3   STH   R5,LOC1-14(R6)    (LOC1) = X*000E*

000034I     5870 4001 0000               37  RX3.EX1   L     R7,Y*10000*       (R7) = (Y*010000*)

00003AI     5885 4601 FFE4               39  RX3.EX2   L     R8,Y*20000*-28(R5,R6) (R8) = (Y*020000*)
000040I     4300 FFBC =000000I           40            B     SF

000044I                                  42            END

 LOCATION   OBJECT INFORMATION    STATEMENT  LABEL   OP-CODE   OPERANDS        COMMENTS
 COUNTER                          NUMBER
```

Figure 1-5 Sample Program

1.8.4 Register-to-Register (RR) Format

561



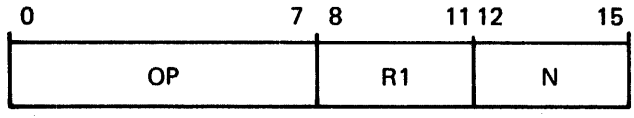
In this 16-bit format, bits 0:7 contain the operation code; bits 8:11 contain the R1 field; and bits 12:15 contain the R2 field. In most RR instructions, the register specified by R1 contains the first operand, and the register specified by R2 contains the second operand. For example:

Machine Code	Label	Assembler Notation
0865	RR	LR R6,R5

- 5 — Second operand
- 6 — First operand
- 08 — Load Register (LR) instruction op-code

1.8.5 Short Form (SF) Format

562



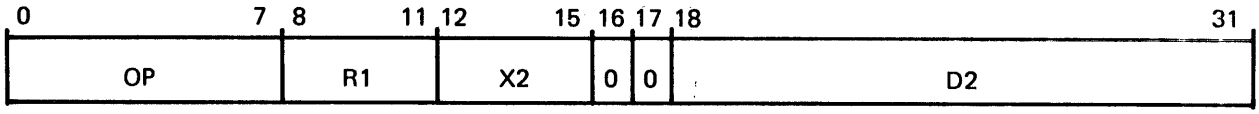
This 16-bit format provides space economy when working with small values. Bits 0:7 contain the operation code; bits 8:11 contain the R1 field; and bits 12:15 contain the N field. In arithmetic and logical operations, the register specified by R1 contains the first cperand. The N field contains a 4-bit immediate value (12:15) used as the second operand. For example:

Machine Code	Label	Assembler Notation
245E	SF	LIS R5,14

- E — Second operand
- 5 — First operand
- 24 — Load Immediate Short (LIS) instruction op-code

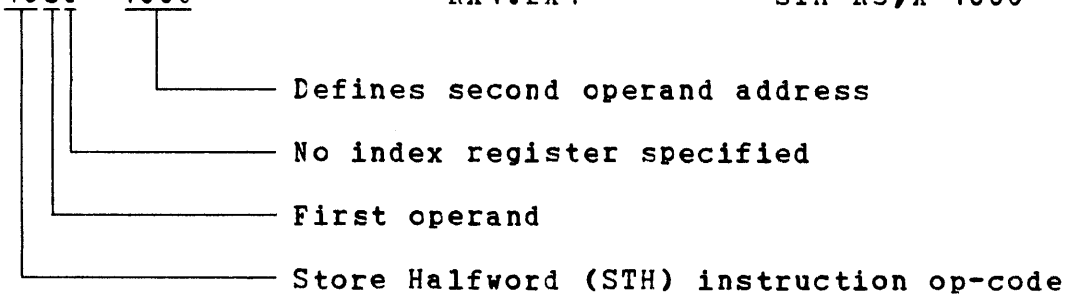
1.8.6 Register and Indexed Storage One (RX1) Format

563



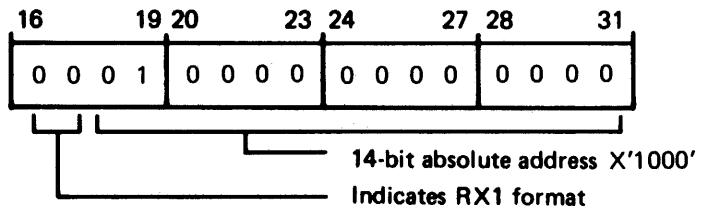
This is a 32-bit format in which bits 0:7 contain the operation code; bits 8:11 contain the R1 field; bits 12:15 contain the X2 field; bits 16 and 17 must be zero; and bits 18:31 contain the D2 field. In general, the register specified by R1 contains the first operand. The second operand is located in memory at the address obtained by adding the contents of the second operand index register (specified by X2) and the 14-bit absolute address contained in the D2 field. For example:

Machine Code	Label	Assembler Notation
4050 1000	RX1.EX1	STH R5,X'1000'



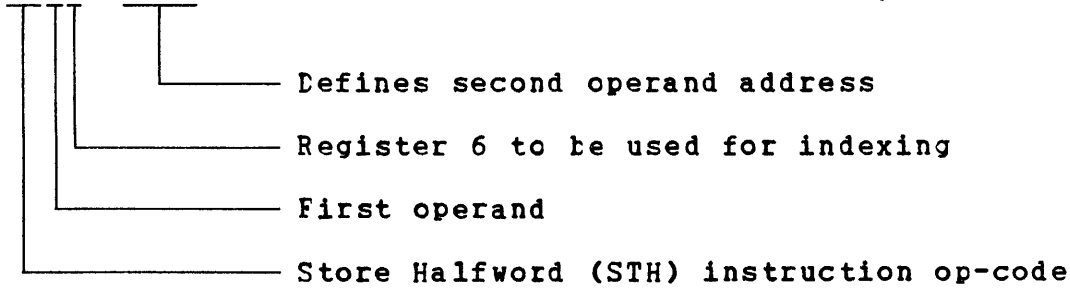
The second operand address is calculated as follows:

564-1

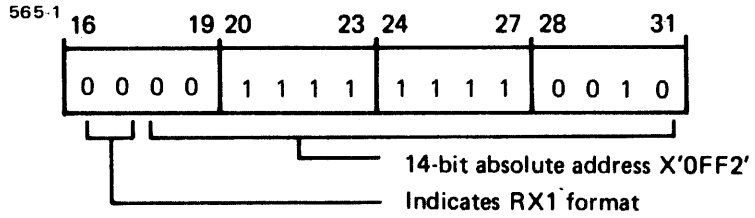


No indexing is specified; therefore, the second operand address is X'1000'.

Machine Code	Label	Assembler Notation
4056 0FF2	RX1.EX2	STH R5,X'0FF2' (R6)



The second operand address is calculated as follows:



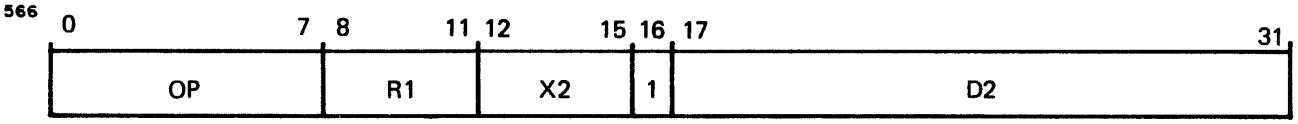
Second Operand Address

= contents of D2 field + contents of index register 6 (see Figure 1-5)

= X'0FF2' + Y'0000000E'

= Y'00001000'

1.8.7 Register and Indexed Storage Two (RX2) Format



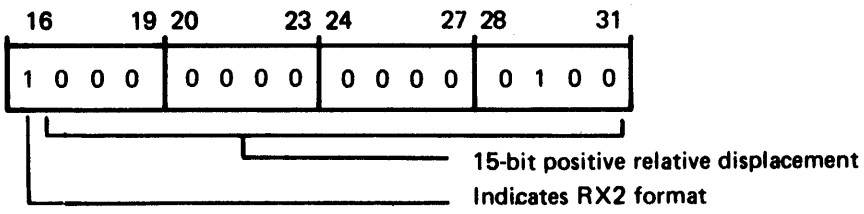
This format provides relative addressing capability in a 32-bit instruction word. Bits 0:7 contain the operand code; bits 8:11 contain the R1 specification; bits 12:15 contain the X2 specification; bit 16 must always be one; and bits 17:31 contain the relative displacement, D2.

In the RX2 format, the register specified by R1 contains the first operand. The address of the second operand, in memory, is calculated by adding the value contained in the incremented location counter (the address of the next sequential instruction) and the sum of (1) the 32-bit representation of the 15-bit signed number contained in the D2 field, and (2) the contents of the index register specified by X2. Negative numbers in the D2 field are expressed in two's complement notation. For example:

Machine Code	Label	Assembler Notation
4050 8004	RX2.EX1	STH R5,LOC1

- 8004 — Defines second operand address
- 50 — No index register specified
- 40 — First operand
- 40 — Store Halfword (STH) instruction op-code

The second operand address is calculated as follows:



Second Operand Address

= 32-bit expansion of contents of D2 field + contents of incremented location counter (see Figure 1-5)

= Y'00000004' + Y'00000010'

= Y'00000014'

Machine Code	Label	Assembler Notation
4050 FFE4	RX2.EX2	STH R5,LOC1

Defines second operand address

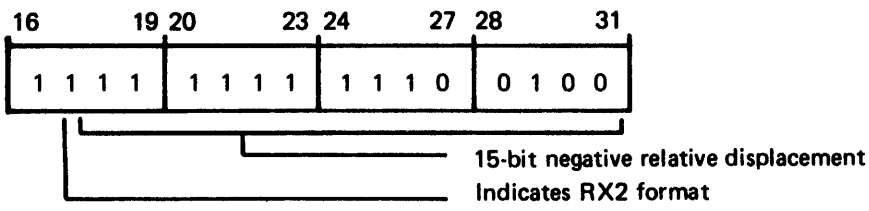
No index register specified

First operand

Store Halfword (STH) instruction op-code

The second operand address is calculated as follows:

568-1



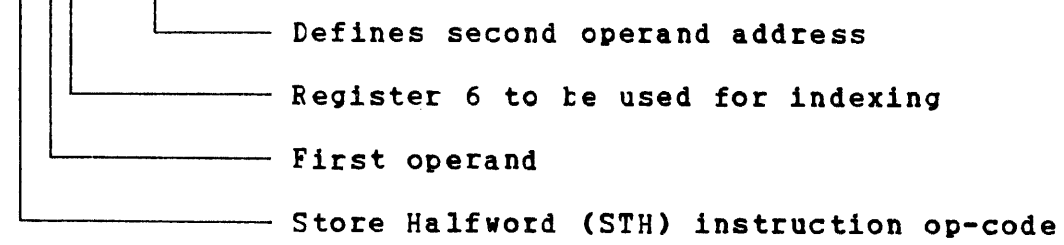
Second Cperand Address

= 32-bit expansion of contents of D2 field + contents of incremented location counter (see Figure 1-5).

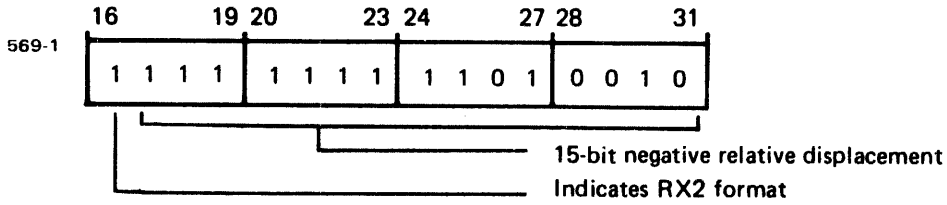
= Y'FFFFFFE4' + Y'00000030'

= Y'00000014'

Machine Code	Label	Assembler Notation
4056 FFD2	RX2.EX3	STH R5,LOC1-14 (R6)



The second operand address is calculated as follows:



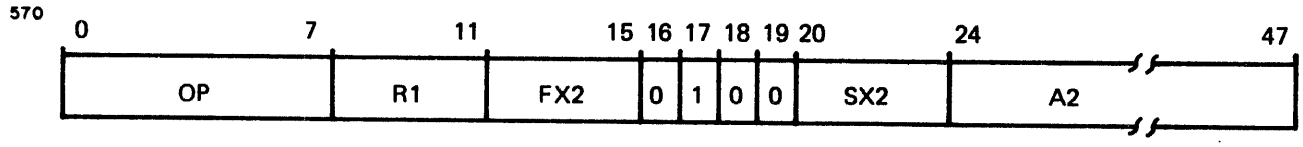
Second Operand Address

= 32-bit expansion of D2 field + contents of incremented location counter + contents of index register 6 (see Figure 1-5)

= Y'FFFFFFD2' + Y'00000034' + Y'0000000E'

= Y'00000014'

1.8.8 Register and Indexed Storage Three (RX3) Format



This is a 48-bit format in which double indexing is permitted. Bits 0:7 contain the operation code; bits 8:11 contain the R1 specification; bits 12:15 contain the first index specification, FX2; bit 16 must be zero; bit 17 must be one; bits 18:19 must be zero; bits 20:23 contain the second index specification, SX2; and bits 24:47 contain a 24-bit address, A2. Second level indexing is allowed even if first level indexing is not specified.

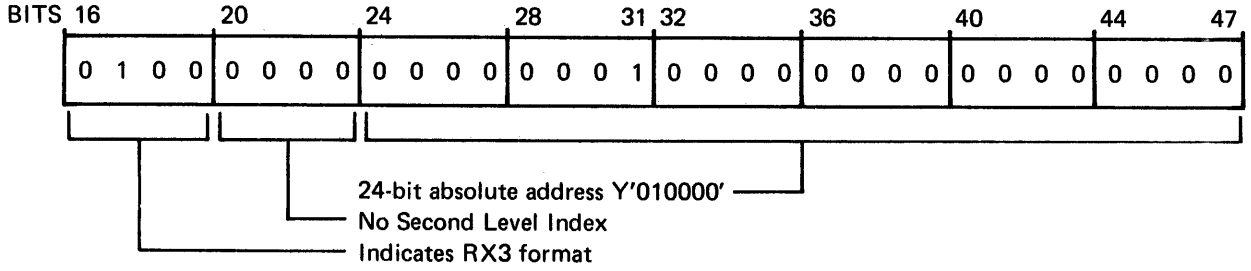
In general, the first operand is contained in the register specified by R1. The second operand is located in memory. Its memory address is obtained by adding the contents of the first index register and the contents of the second index register, and then adding to this result the contents of the A2 field. For example:

Machine Code	Label	Assembler Notation
5870 4001 0000	RX3.EX1	L R7,Y'10000'

- 0000 — Defines second operand address
- 4001 — Second level indexing not specified
- 4001 — Specifies RX3 format
- 5870 — First level indexing not specified
- 5870 — First operand
- 5870 — Load (L) instruction op-code

The second operand address is calculated as follows:

571-1



Second Operand Address

= contents of A2 field

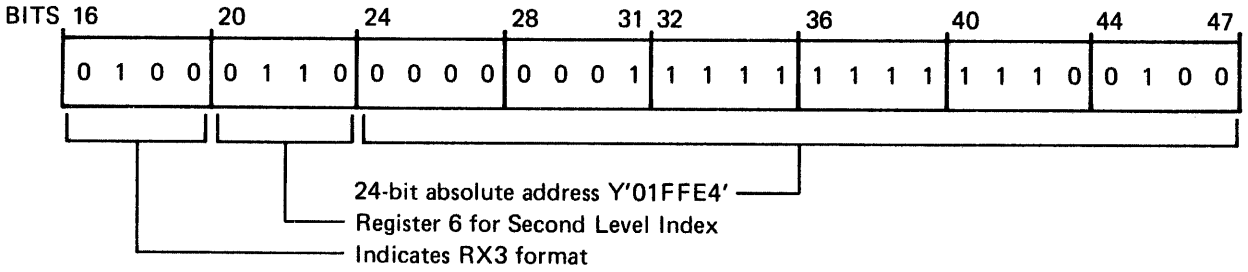
= Y'00010000'

Machine Code	Label	Assembler Notation
5885 4601 FFE4	RX3.EX2	L R8,Y'20000'-28 (R5,R6)

- Defines second operand address
- Register 6 tc be used for second level indexing
- Specifies RX3 format
- Register 5 tc be used for first level indexing
- First operand
- Load (L) instruction op-code

The second operand address is calculated as follows:

572-1



Second Operand Address

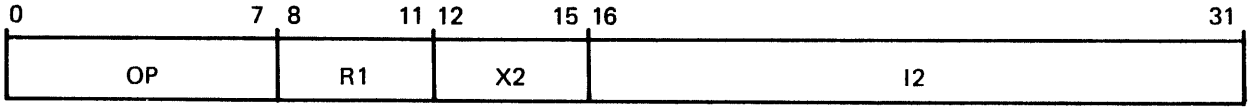
= contents of A2 field + contents of index register 6 + contents of index register 5 (see Figure 1-5)

= Y'0001FFE4' + Y'0000000E' + Y'0000000E'

= Y'00020000'

1.8.9 Register and Immediate Storage One (RI1) Format

573



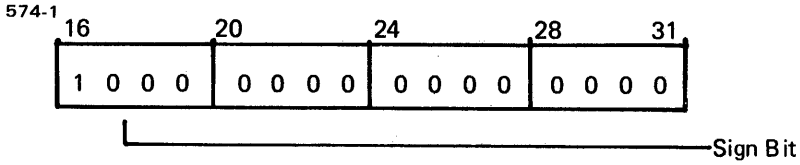
This format represents a 32-bit instruction word. Bits 0:7 contain the operand code; bits 8:11 contain the R1 specification; and bits 16:31 ccntain the 16-bit immediate value, I2.

In this format, the register specified by R1 contains the first operand. The 32-bit effective second operand is obtained by adding together the 32-bit representation of the signed 16-bit value contained in the I2 field, and the contents of the register specified by X2. For example:

Machine Code	Label	Assembler Notation
C890 8000	RI1.EX1	LHI R9,X'8000'

- 16-bit immediate value
- No index register specified
- First operand
- Load Halfword Immediate (LHI) instruction op-code

The second operand is calculated as follows:



Second Operand

= 32-bit representation of X'8000'

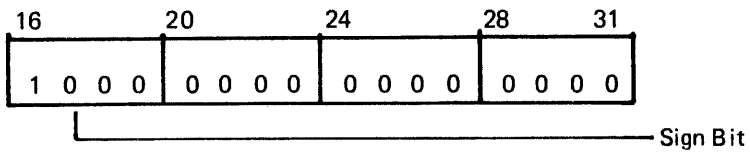
= Y'FFFF8000'

Machine Code	Label	Assembler Notation
C895 8000	RI1.EX2	LHI R9,X'8000'(R5)

- 16-bit immediate value
- Index register 5 specified
- First operand
- Load Halfword Immediate (LHI) instruction op-code

The second operand is calculated as follows:

575-1



Second Operand

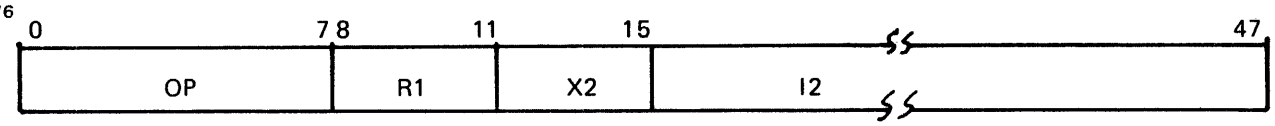
= 32-bit representation cf X'8000' + the contents of the index register 5 (see Figure 1-5)

= Y'FFFF8000' + Y'0000000E'

= Y'FFFF800E'

1.8.10 Register and Immediate Storage Two (RI2) Format

576



This is a 48-bit instruction format. Bits 0:7 contain the operation code; bits 8:11 contain the R1 specification; bits 12:15 contain the X2 specification; and bits 16:47 contain the 32-bit immediate value, I2.

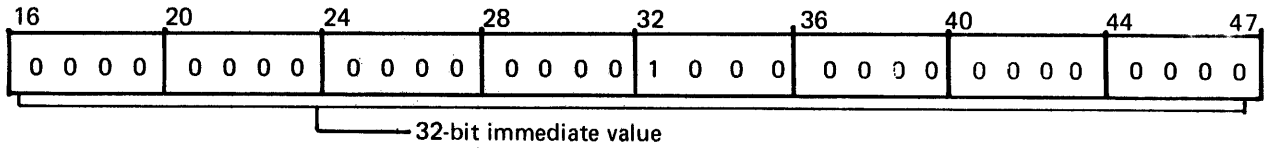
The first operand is contained in the register specified by R1. The second operand is obtained by adding the contents of the index register, specified by X2, and the 32-bit immediate value contained in the I2 field. For example:

Machine Code	Label	Assembler Notation
F8A0 0000 8000	RI2.EX1	LI R10,X'8000'

F8A0 0000 8000

- 32-bit immediate field
- No index register specified
- First operand
- Load Immediate (LI) instruction op-code

The second operand is calculated as follows:



Second Operand

= contents of I2 field

= Y'00008000'

Machine Code	Label	Assembler Notation
F8BA 0001 7FFE	RI2.EX2	LI R11,Y'17FFE' (R10)

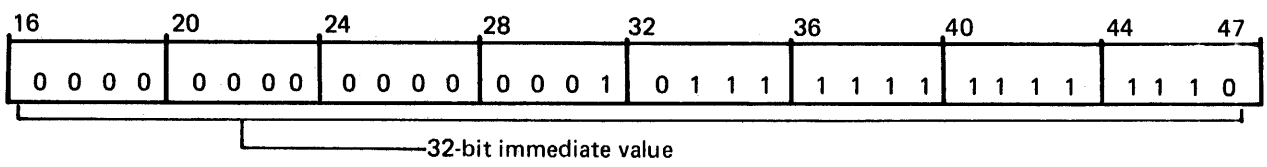
F8BA 0001 7FFE — 32-bit immediate field

BA — Specifies index register 10

B — First operand

F8 — Load Immediate (LI) instruction op-code

The second operand is calculated as follows:



Second Operand

= contents of I2 field + contents of index register 10 (see Figure 1-5)

= Y'00017FFE' + Y'00008000'

= Y'0001FFFE'

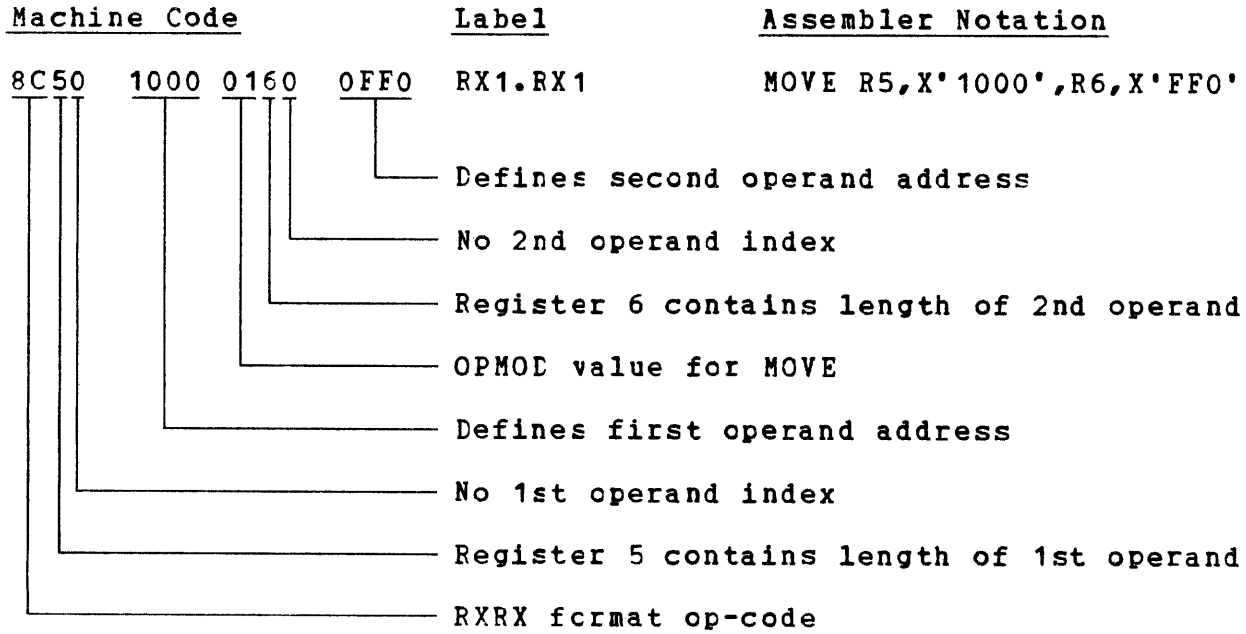
1.8.11 Register and Indexed Storage/Register and Indexed Storage (RXRX) Format (See Figures 1-4 and 1-6)

The RXRX format resembles a pair of adjacent RX format instructions, but represents only one instruction. Each member of the instruction pair may have any one of the standard RX formats. For example, the first member might be RX1 and the second member might be RX3, resulting in a 10-byte instruction. The particular RX format chosen by the assembler for one member is independent of that chosen for the other; thus, the instruction can require 8, 10, or 12 bytes.

OP contains the operation code that defines the RXRX instruction class. The actual operation to be performed is defined by the OPMOD field.

The L1 field specifies the length of the first operand string. If bit 0 of OPMOD is set, L1 is the length with a maximum value of 15. If bit 0 of OPMOD is zero, the general register specified by L1 contains the length. The L2 field specifies the length of the second operand string. If bit 1 of OPMOD is set, this field contains the length with a maximum value of 15. If bit 1 of OPMOD is zero, the general register specified by L2 contains the length.

The effective address calculated for the first member is the address of the left-most (lowest-address) byte of the first operand string. The effective address calculated for the second member is the address of the left-most byte of the second operand string. An RX2 displacement calculated for either member is with respect to the incremented location counter for that member.



In this example both members of the RXRX instruction use the RX1 format. No indexing is specified for either member so the first operand address is X'1000', and the second operand address is X'0FF0'.

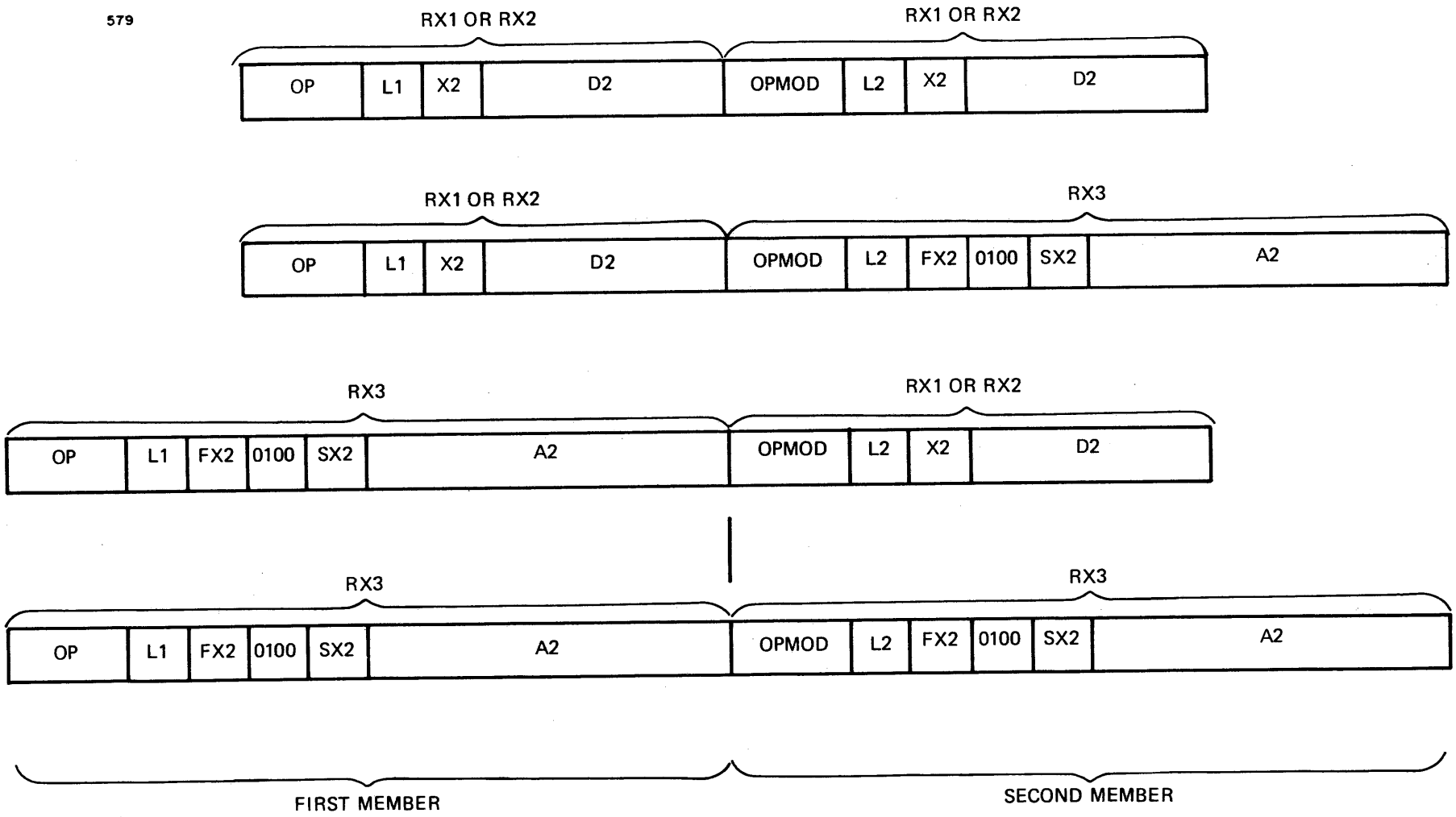
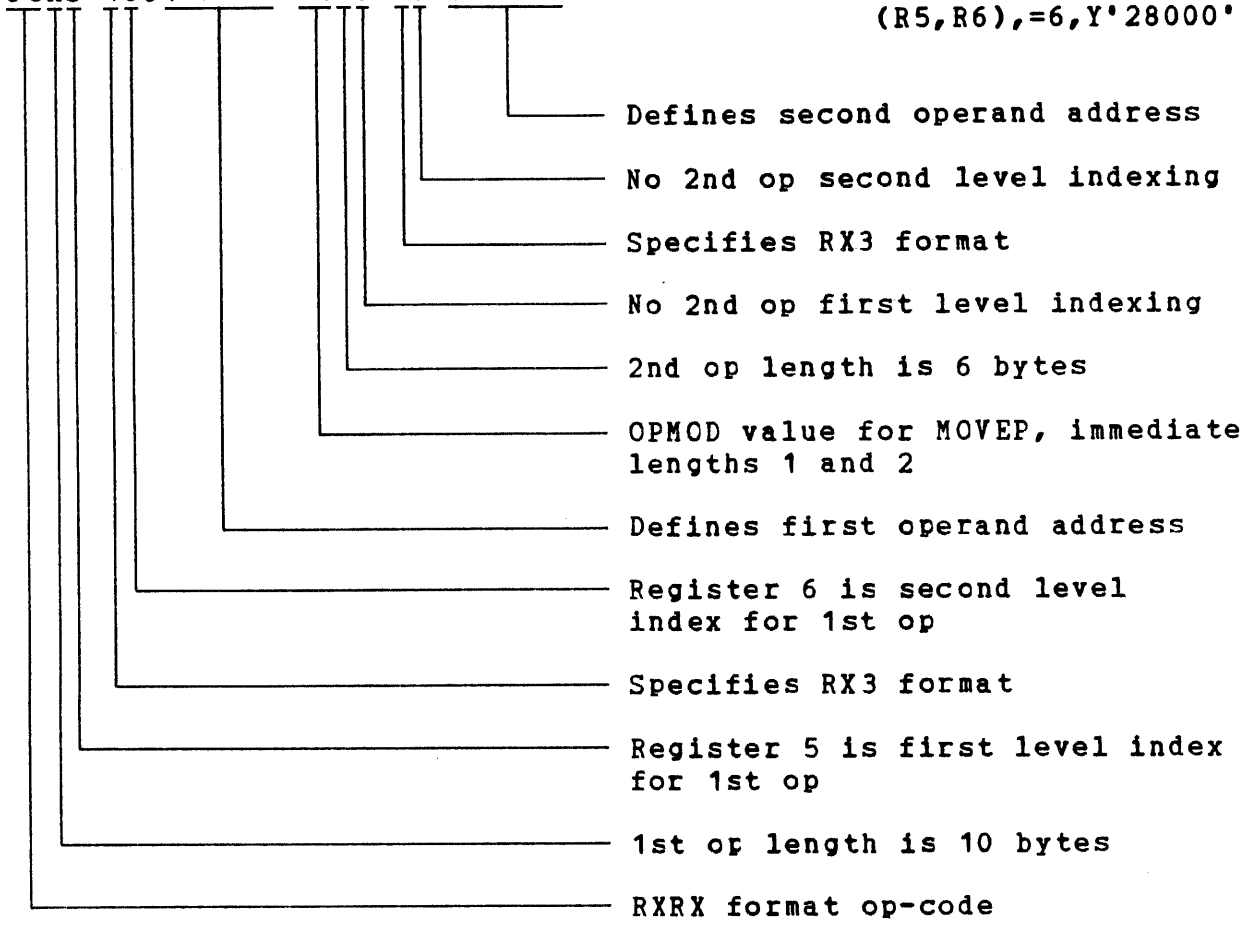


Figure 1-6 RXRX Formats

Machine Code	Label	Assembler Notation
8CA5 4601 FFE4 E160 4002 8000	RX3.RX3	MOVEP =10,Y'1FFE4' (R5,R6),=6,Y'28000'



In this example, both members of the RXRX instruction use the RX3 format. Double indexing is specified for the first member and no indexing is specified for the second member. The first operand address is X'1FFE4' plus the contents of index registers 6 and 5. The second operand address is X'28000'. The length of the first operand is 10 bytes and the second operand is 6 bytes.

CHAPTER 2
SYSTEM CONTROL

2.1 INTRODUCTION

Operator control is provided by the system control panel and the System Terminal, a microcode-supported device interfaced to the system by an asynchronous line controller. The System Terminal may be used as the operating system's console device, and may be a visual display unit or a printing terminal. The asynchronous interface must be strapped as device numbers X'10' and X'11'.

2.2 CONFIGURATION

The system control panel, shown in Figure 2-1, controls power to the system and Initial Program Loading (IPL). It also provides controls for system initialization, processor halt/run, and single step. Light Emitting Diodes (LEDs) on the system console indicate current system state.

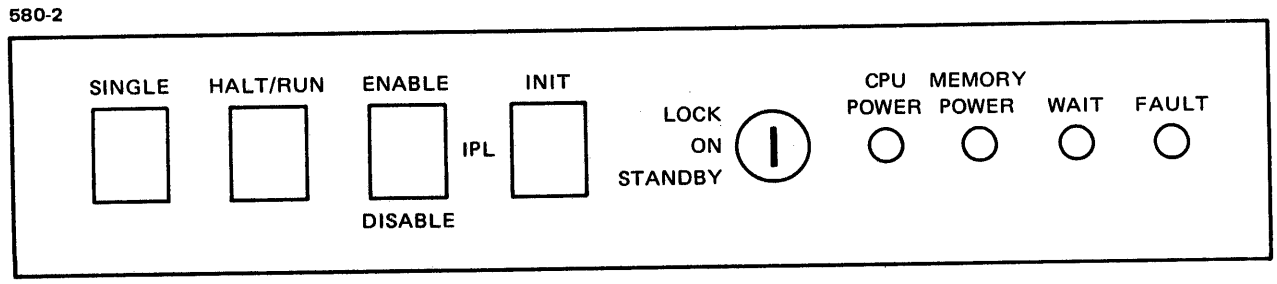


Figure 2-1 System Control Panel

Keyboard commands through the System Terminal allow the operator to examine and modify processor registers and main memory locations and then begin program execution. (Refer to Figure 2-2.) Hexadecimal characters and a number of special characters are recognized by the System Terminal support microcode. The characters accepted and their meanings are shown in Table 2-1. No other characters are accepted and cause a question mark (?) to be written to the System Terminal. When not in use for operator control, the System Terminal is available to a running program for use as an I/O device. See Appendix F for a flowchart of the console service routine.

TABLE 2-1 SYSTEM TERMINAL SUPPORT COMMAND SUMMARY

581-1

KEY COMMAND SEQUENCE	MEANING	SYSTEM TERMINAL DISPLAY
@ n n n n n n CR	Select memory address and display halfword contents	≤@nnnnnn nnnnnn YYYY ≤
R n CR	Select general register and display contents	≤Rn YYYYYYYY ≤
F n CR	Select single-precision floating-point register and display contents*	≤Fn YYYYYYYY ≤
D n CR	Select double-precision floating-point register and display contents*	≤Dn YYYYYYYY YYYYYYYY ≤
P CR	Select program status word and display contents	≤P YYYYYY YYYYYY ≤
+	Increment memory location counter to display next sequential halfword	≤+ nnnnnn YYYY ≤
-	Decrement memory location counter to display previous halfword	≤- nnnnnn YYYY ≤
= Y Y Y Y CR	Replace contents of currently selected memory location or register with new data	≤=YYYY for memory ≤ ≤=YYYYYYYY for register ≤
<	Begin program execution at current memory location	≤<
#	Delete command	≤@10# ≤
>	Single-step instruction at current memory location	≤>

*Floating-point is optional in this processor.

Notes:

1. Characters in boxes indicate operational key strokes required for commands.

2. Character symbol cf lower case n used to indicate hexadecimal address of memory or register.

3. Character symbol cf upper case Y used to indicate hexadecimal contents of memory or register.

4. Underlined characters are those output from the system. Characters not underlined are those typed by the operator.

5. A back arrow, or underline (X'5F'), or a back space (X'08') character may be used to delete the previously input hexadecimal character.

6. Space characters may be entered as desired. They are ignored by the processor.

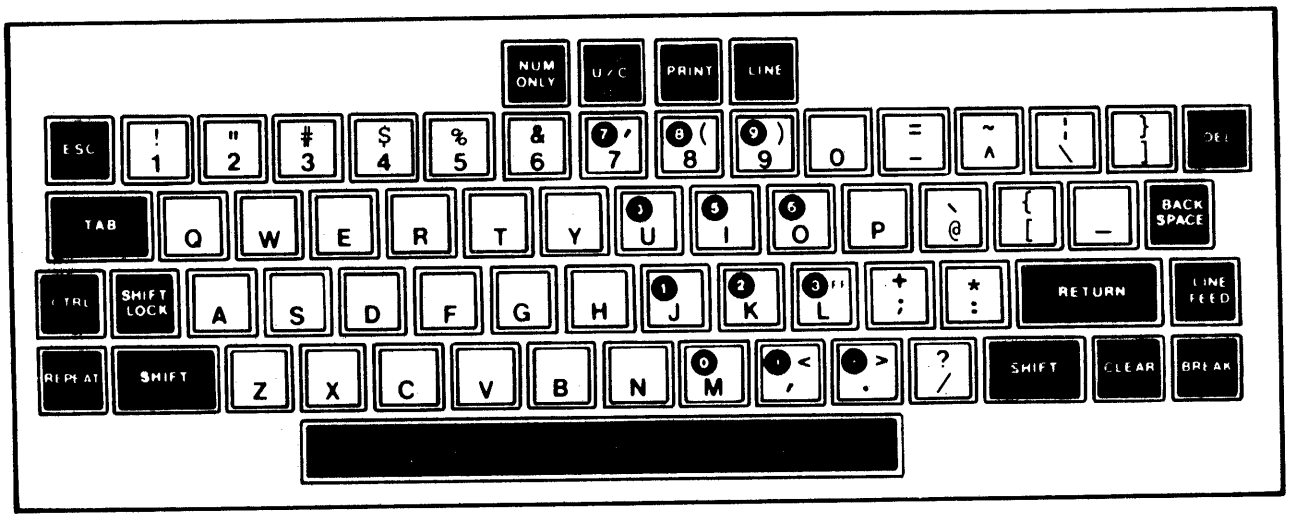


Figure 2-2 Model 550 Keyboard Layout

2.3 SYSTEM CONTROL PANEL SWITCHES AND INDICATORS

2.3.1 Key-Operated Security Lock

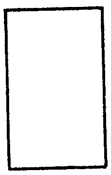
This is a 3-position (STANDBY-ON-LOCK) key-operated switch that controls primary power to the system. It can also disable (LOCK) the initialize and console switches, thereby preventing any accidental manual input to the system. The power indicator lamp (POWER) is on when the security lock is in the ON or LOCK position.

2.3.2 Control Switches

All the control switches, with the exception of the Initial Program Load (IPL) switch, are enabled only when the key-operated security lock is in the CN position, and primary AC power is applied.

HALT/RUN

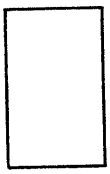
HALT/RUN



This momentary contact switch causes program execution to be halted if the system was running, or resumed if the system was halted. When halted, control is given to the System Terminal support routine through which the memory or registers can be examined or modified and program execution restarted. If the processor was already in the System Terminal support routine, program execution is started. This switch is disabled if the security lock is in the LOCK position.

SINGLE STEP

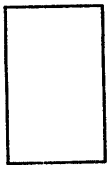
SINGLE



When in the UP position, control is automatically given to the System Terminal support routine at the conclusion of each user level instruction. The program status word is displayed, including the address of the next sequential instruction (location counter). Execution of the next instruction is caused by pressing the HALT/RUN switch or by typing a less than (<) character on the System Terminal. To resume normal run mode execution, return the SINGLE STEP switch to the DOWN position and begin execution by pressing the HALT/RUN switch or by typing the less than (<) character on the System Terminal. The SINGLE STEP switch is disabled when the security lock is in the LOCK position. Attempts to single step through instructions that do I/O to the System Terminal do not produce meaningful results.

IPL

ENABLE



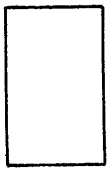
DISABLE

This switch is not disabled by the security lock. When in the ENABLE position, an Initial Program Load (IPL) from the Loader Storage Unit (LSU) is performed after any of the following steps:

1. turning the security lock from the STANDBY to ON position
2. depression of the Initialize (INIT) switch
3. return of AC power to the system

INITIALIZE

INIT



This momentary contact switch causes the system to be initialized. The initialization sequence clears all device controllers on the I/O bus and resets certain functions in the processor. The fault lamp (FAULT) comes on when the switch is depressed and is extinguished with the completion of the initialization sequence.

2.4 OPERATING INSTRUCTIONS

2.4.1 Power-Up

To prevent Initial Program Load (IPL) on power-up, place the IPL switch in the DISABLE position. To power-up the system, turn the key-operated security lock clockwise from the STANDBY to the ON position. The power lamp (POWER) lights, and power is provided to the system. The fault lamp (FAULT) on the system control panel also lights, and the microdiagnostic routine is entered. This routine exercises internal data paths and registers. If main memory power has fallen out of regulation since the system was last running, locations X'000000' to X'03FFFF' are initialized. The diagnostic routine tests the lowest 256 kbytes of memory before extinguishing the FAULT lamp. This diagnostic is limited in scope, serving only to indicate a go/no-go condition. If an error is detected in any portion of the microdiagnostic, the microccde loops indefinitely, and the FAULT lamp remains on. If no errors are detected, the FAULT lamp is turned off.

2.4.2 Entering Console Service

If power was lost while the microcode was in the console service routine, control is returned to the console when the power-up sequence is complete, provided that IPL is not enabled. If the system was executing a program when power was lost, execution resumes when power returns, provided that IPL is not enabled. To enter console service in this case, depress the HALT/RUN switch.

2.4.3 Initial Program Load (IPL)

To perform Initial Program Load (IPL), place the IPL switch in the ENABLE position; then initialize the system by depressing the INIT switch momentarily. A power down/power up sequence is emulated, and diagnostics are performed. At the successful completion of the microdiagnostic sequence, an IPL from the LSU is performed. Control is transferred to the newly-loaded program.

2.5 SYSTEM TERMINAL COMMANDS

When the System Terminal support routine is entered from power-up or initialize, a carriage return and line feed sequence are output. The current value of the PSW status and location counter are output, followed by another carriage return and line feed sequence. Finally, the less than (<) operator prompt character is output to indicate that the system is ready to receive operator commands. If memory power was lost, the location counter is set to X'003FFFFE', and the PSW is set to X'00008000'. In this case, the first 256 kbytes of memory are written during power-up to establish the error correcting code bits.

Space characters may be used as desired in any of the described System Terminal commands. Spaces are ignored by the console routine.

2.5.1 Select an Address and Examine "@"

The commercial "at" sign (@) places the System Terminal support routine in the address mode. This character may be followed by up to six hexadecimal digits of address. Leading zeros are not required. If more than six digits are input, only the least significant six are used. A carriage return is used to signal the end of the address; the address input is then copied into the location counter. A carriage return and line feed sequence are output, followed by the new value of the location counter and the halfword contents of that location. Note that the data fetch is subject to memory relocation if enabled by the current PSW. After this display, a carriage return and line feed sequence are output, followed by a new operator prompt.

If an invalid character is input by the operator, the system responds by outputting a question mark (?), a carriage return, a line feed, and an operator prompt.

2.5.2 Increment and Examine Next Location "+"

After examining a memory location, the plus character (+) can be used to advance the location counter by two. No other operator input is required. A carriage return and line feed are output, followed by the new location counter value and the halfword contents of that location. This memory access is subject to the relocation defined by the current PSW. After outputting another carriage return and line feed, the operator prompt character is output. This procedure may be repeated to examine sequential memory locations.

2.5.3 Decrement and Examine Prior Location "-"

After examining a memory location, the minus character (-) can be used to decrement the location counter by two. No other operation is required. A carriage return and line feed are output, followed by the new location counter value and the halfword contents of that location. This memory access is subject to the relocation defined by the current PSW. After outputting another carriage return and line feed, the operator prompt character is output. This procedure may be repeated to examine sequential memory locations.

2.5.4 Modify Current Location "="

After examining a memory location, the equal sign (=) can be used to put the System Terminal support routine in the memory write mode. This character may be followed by up to four hexadecimal digits of data to be written. Leading zeros are not required. If more than four digits are input, only the least significant four are used. A carriage return is used to signal the end of the data. At that time, the accumulated data is written into the memory location currently addressed by the location counter. This memory write is subject to the relocation defined by the current PSW. The current location counter is incremented by two and a carriage return, line feed, and operator prompt are output. This procedure may be repeated to modify sequential memory locations.

2.5.5 Examine General Register "R"

The character (R) causes the System Terminal support routine to interpret subsequent hexadecimal input as the number of a general register (in the set selected by the current PSW) to be displayed. A carriage return is used to signal the end of hexadecimal input. At that time, the least significant four bits of the accumulated hexadecimal data are taken as the desired register number. The fullword contents of that register are output followed by a carriage return, line feed, and operator prompt. Plus and minus commands are invalid for general registers.

2.5.6 Modify General Register "="

Immediately after examining a general register, the equal sign (=) can be used to change the contents of the currently selected register. The equal sign can be followed by up to eight hexadecimal digits of data. Leading zeros are not required. If more than eight digits are input, only the least significant eight are used. A carriage return is used to signal the end of the data input. At that time, the accumulated data is copied into the currently selected general register. A carriage return, line feed, and operator prompt are then output.

2.5.7 Examine Single-Precision Floating-Point Register "F"

The character (F) causes the System Terminal support routine to interpret subsequent hexadecimal input as the number of a single-precision floating-point register to be displayed. If the processor does not have single-precision floating-point, this command character causes a question mark sequence to be output. A carriage return is used to signal the end of hexadecimal input. At that time, the least significant four bits of the accumulated hexadecimal data are taken as the desired register number. If necessary, this number is rounded to the next lowest even number. The fullword contents of that register are output followed by a carriage return, line feed, and operator prompt. Plus and minus commands are invalid for flcating-point registers.

2.5.8 Modify Single-Precision Floating-Point Register "="

Immediately after examining a single-precision floating-point register, that register is available for modification. Type an equal sign (=) followed by up to eight hexadecimal digits of data. Leading zeros are not required. If more than eight digits are input, only the least significant eight are used. A carriage return is used to signal the end of the data input. At that time, the accumulated data is copied into the currently selected single-precision floating-point register. This data is not tested for normalization; therefore, an unnormalized floating-point number can be manually placed in the register. The system outputs a carriage return, line feed, and operator prompt.

2.5.9 Examine Double-Precision Floating-Point Register "D"

The character (D) causes the System Terminal support routine to interpret subsequent hexadecimal input as the number of a double-precision floating-point register to be displayed. If the processor does not have double-precision floating-point, this command character causes a question mark sequence to be output. A carriage return is used tc signal the end of hexadecimal input. At that time, the least significant four bits of the accumulated hexadecimal data are taken as the desired register number. If necessary, this number is rounded to the next lowest even number. The doubleword contents of that register are output, followed by a carriage return, line feed, and operator prompt. Plus and minus commands are invalid for floating-point registers.

2.5.10 Modify Double-Precisicn Floating-Point Register "="

Immediately after examining a double-precision floating-point register, that register is available for modification. Type an equal sign (=) followed by up to 16 hexadecimal digits. Leading zeros are not required. If more than 16 digits are input, only the last 16 digits are used. A carriage return is used to signal the end of the data input. At that time, the accumulated data is copied into the currently selected double-precision register. The data is not tested for normalization; therefore, an unnormalized floating-point number can be manually placed in a double-precision register. The system outputs a carriage return, line feed, and operator prompt.

2.5.11 Examine Program Status Word "P"

The character (P) puts the System Terminal support routine into the PSW display mode. A carriage return is required to complete this command input. Upon receipt of the carriage return, the contents of the PSW are output followed by a carriage return, line feed, and operator prompt. The plus and minus commands are invalid for the PSW.

2.5.12 Modify Program Status Word "="

Immediately after examining the PSW, the equal sign (=) can be used to change the contents of the PSW status field. The equal sign can be followed by up to six hexadecimal digits of data. Leading zeros are not required. If more than six digits are input, only the least significant six are used. A carriage return is used to signal the end of the data input. At that time, the accumulated data is copied into the PSW, which is then displayed. A carriage return, line feed, and operator prompt are then output.

2.5.13 Execute Single Instruction ">"

Entering the character (>) causes the processor to execute the instruction indicated by the location counter, in single-step mode. If the instruction is an interruptible instruction, only one phase or iteration of the instruction may be performed. After this execution, the ccnsole service routine displays the PSW and location counter, followed by a carriage return, line feed, and operator prompt.

2.5.14 Enter Run Mode "<"

Entering the character (<) causes the processor to begin program execution, starting with the instruction indicated by the location counter.

2.6 MEMORY INITIALIZATION

The following example shows how to set up dedicated low memory for loading the 32-bit relocating loader from magnetic tape.

≤ [@] [3] [0] [cr]	Select address '30'
000030 0000	Location '30' already = '0000'
≤ [+]	Advance to address '32'
000032 8000	Location '32' already = '8000'
≤ [+]	Advance to address '34'
000034 0000	Location '34' already = '0000'
≤ [+]	Advance to address '36'
000036 1536	Location '36' contains '1536'
≤ [=] [5] [0] [cr]	Change contents of '36' to '0050'
000038 0000	Location '38' contains '0000'
≤ [@] [5] [0] [cr]	Select address '50'
000050 D500	Location '50' already = 'D500', the auto-load instruction
≤ [+]	Advance to address '52'
000052 00CF	Location '52' already = '00CF', the usual ending address
≤ [+]	Advance to address '54'
000054 4300	Location '54' already = '4300' a branch instruction
≤ [+]	Advance to address '56'
000056 0080	Location '56' already = '0080' the usual branch address
≤ [@] [7] [8] [cr]	Select address '78'
000078 C186	Location '78' contains 'C186'

≤ = [8] [5] [A] [1] [cr]	Change '78' to '85A1', the device number and command byte for magnetic tape
00007A 0000	Location '7A' contains '0000'
≤ [+]	Advance to address '7C'
00007C 0000	Location '7C' contains '0000'
≤ [@] [3] [0] [cr]	Select starting address '30'
000030 0000	
≤ [<]	Start program execution

After loading, the relocating loader places the processor in the wait state. The wait lamp on the consolette is on. Depress the HALT/RUN switch to regain control at the System Terminal. The terminal response, for example is:

```
008000 03FB00
<
```

which shows the PSW and the LCC pointing at the loader start address of '3FB00'. Type the less than (<) character to begin execution of the relocating loader.

2.7 SYSTEM TERMINAL PROGRAMMING INSTRUCTIONS

The System Terminal (ST) uses either a 2-line asynchronous communication multiplexor or an 8-line asynchronous communication multiplexor interface. Since the microprogram of the processor must communicate with the ST, the device address is fixed at X'010' and X'011'. The interface must be strapped for full duplex operation. Refer to the appropriate instruction manual for complete programming information.

The microprogram programs the ST for highest clock rate, two stop bits per character, seven data bits, and even parity. Echoplex is not turned on.

CHAPTER 3
LOGICAL OPERATIONS

3.1 INTRODUCTION

The set of logical instructions provides a means for the manipulation of binary data. Many of the instructions grouped with the logical set may also be used in arithmetic and other operations. These instructions include loads, stores, compares, shifts, list processing, translation, and cyclic redundancy checks.

3.2 LOGICAL DATA FORMATS

Logical data may be organized as bytes, halfwords, fullwords, or bit arrays of up to 2^{27} bits as shown in Figure 3-1.

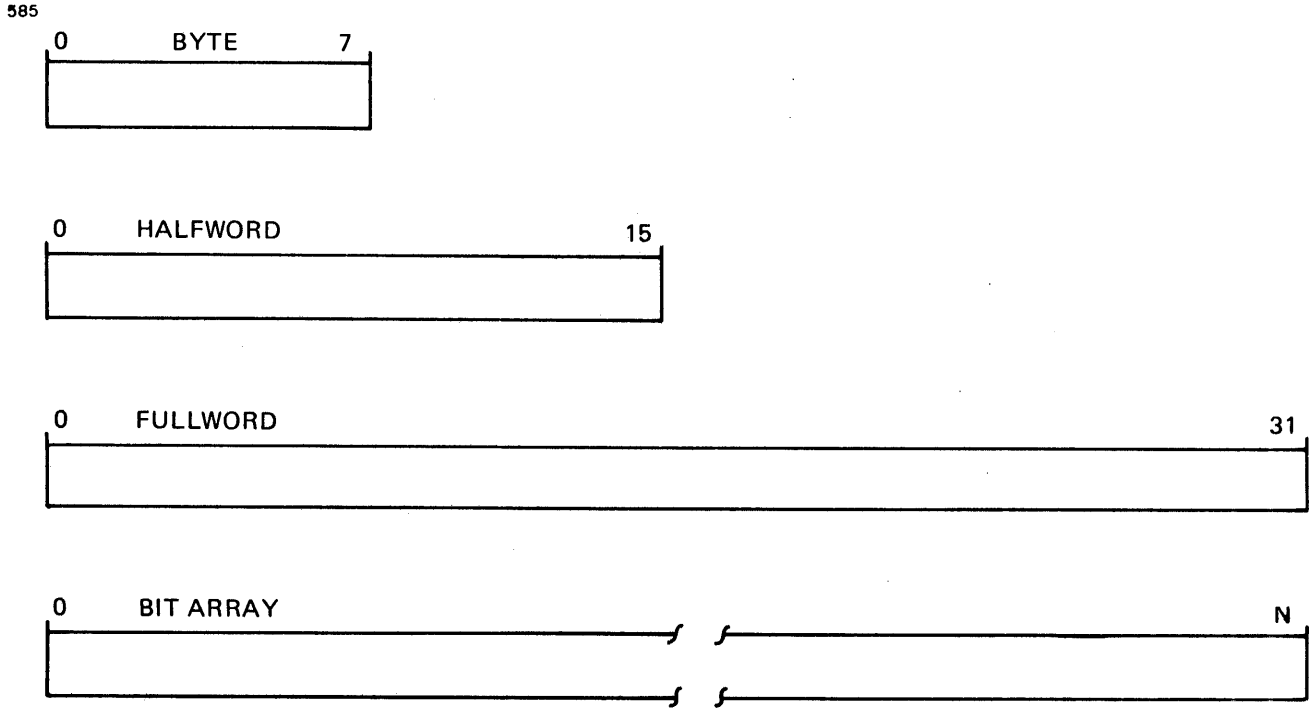


Figure 3-1 Logical Data

3.3 OPERATIONS

In logical operations between the contents of a general register and a halfword operand, the halfword operand is expanded to a fullword before the operation starts. The halfword is expanded by propagating the most significant bits through bits 0:15 of the fullword. For example, the halfword 'A000' is expanded to 'FFFFA000' before participating in the operation.

3.3.1 Boolean Operations

The Boolean operators AND, OR, and Exclusive OR (XOR) operate on halfword and fullword quantities. All bits in both operands participate individually. The Boolean functions are defined as follows:

```
0 AND 0 = 0
0 AND 1 = 0      (logical product)
1 AND 0 = 0
1 AND 1 = 1

0 CR 0 = 0
0 OR 1 = 1       (logical sum)
1 OR 0 = 1
1 OR 1 = 1

0 XCR 0 = 0
0 XOR 1 = 1      (logical difference)
1 XOR 0 = 1
1 XCR 1 = 0
```

3.3.2 Translation

The Translate (TLATE) instruction is used to translate a character directly, or tc effect an unconditional branch to a special translate subroutine. Associated with the translate instruction is a translation table. The entries in the table are halfwords, as shown in Figure 3-2.

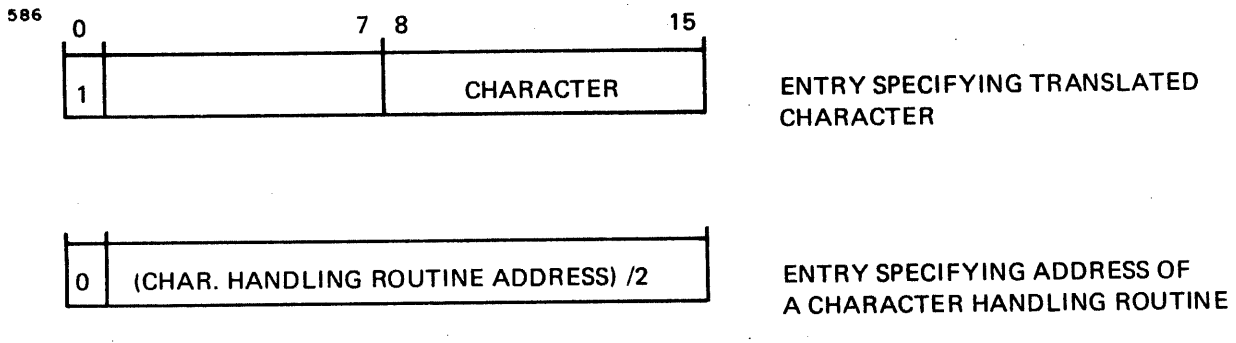


Figure 3-2 Translation Table Entry

The character to be translated is a byte of logical data. This unsigned quantity is doubled and used as an index into the translation table. If the corresponding table entry has a one in bit position zero, then bits 8:15 contain the character to be substituted for the data character. If there is a zero in bit position zero, bits 1:15 contain the address, divided by two, of the translation routine. When the translate instruction results in a branch, this value is doubled to produce the address of the routine. Because this result is a 16-bit address, the software routine must be located in the first 64 kb of the program address space. The program may reside anywhere in memory if it is relocated by the Memory Address Translator (MAT). The translation table may contain up to 256 entries. However, if the data characters are always less than eight bits, fewer entries are required.

3.3.3 List Processing

The list processing instructions manipulate a circular list as defined in Figure 3-3.

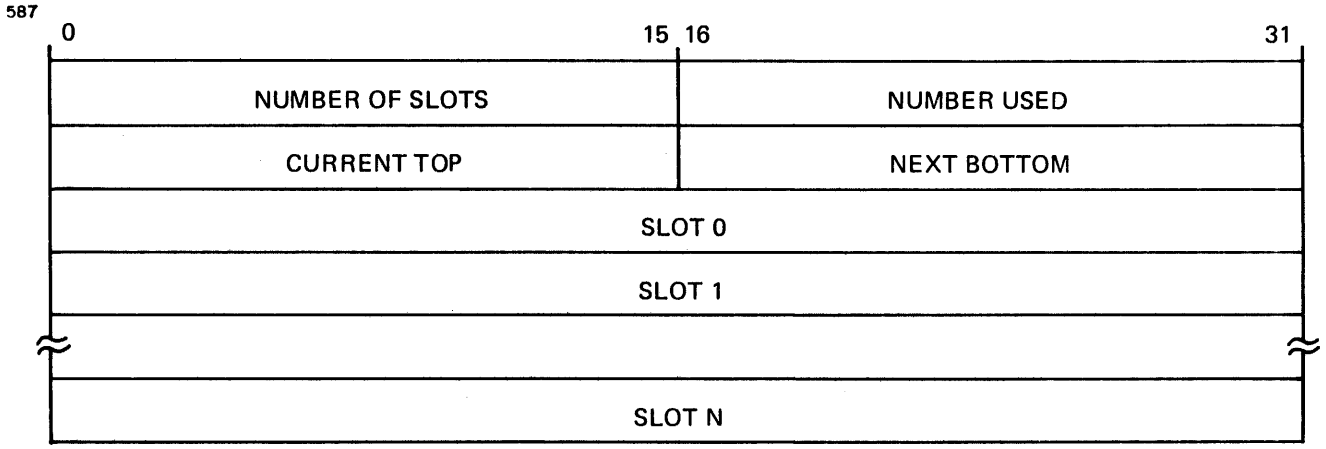


Figure 3-3 Circular List Definition

The first four halfwords, called the list header, contain the list parameters. Immediately following the header is the list itself. The first fullword in the list is designated Slot 0. The remaining slots are designated 1, 2, 3, etc., up to a maximum slot number, which is equal tc the number in the list minus one. An absolute maximum of 65,535 fullword slots may be specified. (Slots are designated 0 through X'FFFE'.)

The first halfword of the header indicates the number of slots (fullwords) in the entire list. The second halfword indicates the current number of slots being used. When this halfword equals zero, the list is empty. When this halfword equals the number of slots in the list, the list is full. Once initialized, this halfword is maintained automatically. It is incremented when elements are added to the list and decremented when elements are removed.

The third and fourth halfwords of the list header specify the current top of the list and the next bottom of the list, respectively. These pointers are also updated automatically. (See Figure 3-4.)

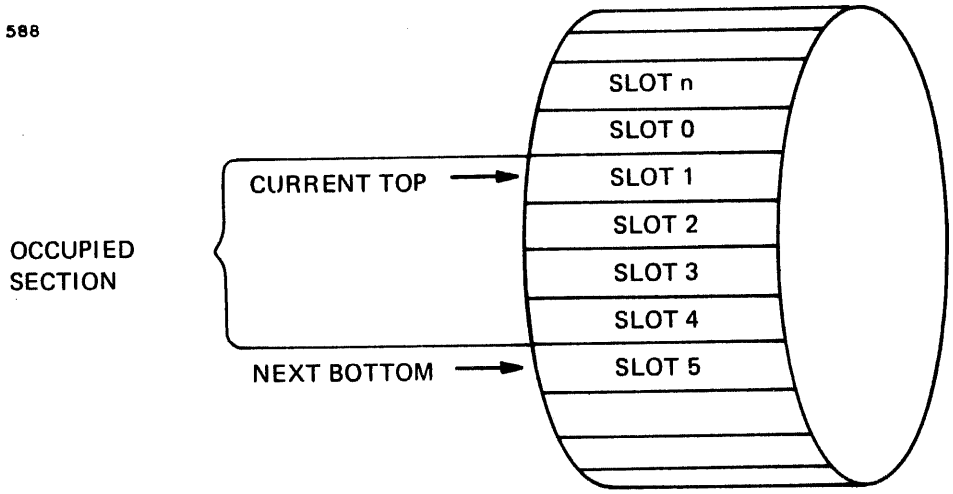


Figure 3-4 Circular List

3.4 LOGICAL INSTRUCTION FORMATS

The logical instructions use the Register to Register (RR), the Short Form (SF), the Register and Indexed Storage (RX), and the Register and Immediate Storage (RI) instruction formats.

3.5 LOGICAL INSTRUCTIONS

The instructions described in this section are:

L	Load
LR	Load Register
LI	Load Immediate
LIS	Load Immediate Short
LCS	Load Complement Short
LH	Load Halfword
LHI	Load Halfword Immediate
LA	Load Address
LRA	Load Real Address
LHL	Load Halfword Logical
LM	Load Multiple
LB	Load Byte
LBR	Load Byte Register
EXHR	Exchange Halfword Register
EXBR	Exchange Byte Register
ST	Store
STH	Store Halfword
STM	Store Multiple
STB	Store Byte
STBR	Store Byte Register
CL	Compare Logical
CLR	Compare Logical Register
CLI	Compare Logical Immediate
CLH	Compare Logical Halfword
CLHI	Compare Logical Halfword Immediate
CLB	Compare Logical Byte
N	AND
NR	AND Register
NI	AND Immediate
NH	AND Halfword
NHI	AND Halfword Immediate
O	OR
OR	OR Register
OI	OR Immediate
OH	OR Halfword
OHI	OR Halfword Immediate
X	Exclusive OR
XR	Exclusive OR Register
XI	Exclusive OR Immediate
XH	Exclusive OR Halfword
XHI	Exclusive OR Halfwcrd Immediate
TI	Test Immediate
THI	Test Halfword Immediate

SLL	Shift Left Logical
SLLS	Shift Left Logical Short
SRL	Shift Right Logical
SRLS	Shift Right Logical Short
SLHL	Shift Left Halfword Logical
SLHLS	Shift Left Halfword Logical Short
SRHL	Shift Right Halfword Logical
SRHLS	Shift Right Halfword Logical Short
RLL	Rotate Left Logical
RRL	Rotate Right Logical
TS	Test and Set
TBT	Test Bit
SBT	Set Bit
CBT	Complement Bit
RBT	Reset Bit
CRC12	Cyclic Redundancy Check Modulo 12
CRC16	Cyclic Redundancy Check Modulo 16
TLATE	Translate
ATL	Add to Top of List
ABL	Add to Bottom of List
RTL	Remove from Top of List
RBL	Remove from Bottom of List

3.5.1 Load (L, LR, LI)

Load (L)
Load Register (LR)
Load Immediate (LI)

Assembler Notation		Op-Code	Format
L	R1,D2(X2)	58	RX1,RX2
L	R1,A2(FX2,SX2)	58	RX3
LR	R1,R2	08	RR
LI	R1,I2(X2)	F8	RI2

Operation

The second operand replaces the contents of the register specified in R1.

Condition Code

C	V	G	L	
0	0	0	0	Value is zero
0	0	0	1	Value is not zero
0	0	1	0	Value is not zero

Programming Notes

When the Load instructions operate on fixed-point data, the condition code indicates zero (no flags), negative (L flag), or positive (G flag) value.

In the RR format, if R1 equals R2, the Load instruction functions as a test on the contents of the register.

In the RX formats, the second operand must be located on a fullword boundary.

3.5.2 Load Immediate Short (LIS)

Assembler Notation	Op-Code	Format
LIS R1,N	24	SF

Operation

The 4-bit second operand is expanded to a 32-bit fullword with high order bits forced to zero. This fullword replaces the contents of the register specified by R1.

Condition Code

C	V	G	L	
0	0	0	0	Value is zero
0	0	1	0	Value is not zero

Programming Note

When this instruction operates on fixed-point data, the condition code indicates zero (no flags), or positive (G flag) value.

Example: LIS

Assembler Notation	Machine Code	Comments
LIS REG4,15	244F	LOAD 15 INTO REG4

Result of LIS Instruction

(REG4) = 0000000F
Condition Code = 0010 (G=2)

3.5.3 Load Complement Short (LCS)

Assembler Notation	Op-Code	Format
LCS R1,N	25	SF

Operation

The 4-bit second operand is expanded to a 32-bit fullword with high order bits forced tc zero. The two's complement value of this fullword then replaces the contents of the register specified by R1.

Condition Code

C	V	G	L	
0	0	0	0	Value is zero
0	0	0	1	Value is not zero

Programming Note

When this instruction operates on fixed-point data, the condition code indicates zero (no flags), or negative (L flag) value.

Example: LCS

Assembler Notation	Machine Code	Comments
LCS REG8,7	2587	LCAD -7 INTO REG8

Result of LCS Instruction

(REG8) = FFFF FFF9
Condition Code = 0001 (L=1)

3.5.4 Load Halfword (LH, LHI)

Load Halfword (LH)
Load Halfword Immediate (LHI)

Assembler Notation		Op-Code	Format
LH	R1,D2(X2)	48	RX1,RX2
LH	R1,A2(FX2,SX2)	48	RX3
LHI	R1,I2(X2)	C8	RI1

Operation

The halfword second operand is expanded to a fullword by propagating the most significant bit through bits 0:15. This fullword replaces the contents of the register specified by R1.

Condition Code

C	V	G	L	
0	0	0	0	Value is zero
0	0	0	1	Value is not zero
0	0	1	0	Value is not zero

Programming Notes

When the Load Halfword instructions operate on fixed-point data, the condition code indicates zero (no flags), negative (L flag), or positive (G flag) value.

In the RX formats, the second operand must be located on a halfword boundary.

In the RI1 format, the 16-bit I2 field is extended to a fullword by propagating the sign bit through bits 0:15. The contents of the index register specified by X2 are then added to form the fullword second operand.

3.5.5 Load Address (LA)

Assembler Notation	Op-Code	Format
LA R1,D2(X2)	E6	RX1,RX2
LA R1,A2(FX2,SX2)	E6	RX3

Operation

The effective address of the second operand (24 bits) replaces bits 8:31 of the register specified by R1. Bits 0:7 of the register specified by R1 are forced to zero.

Condition Code

Unchanged

Programming Note

The length of the address quantity depends on the internal structure of the particular machine; thus, in this processor, with a maximum address length of 24 bits, the calculated address replaces bits 8:31 of the register specified by R1, and bits 0:7 are replaced by zero. In a processor with a maximum address length of 20 bits, the calculated address replaces bits 12:31 of the register specified by R1, and bits 0:11 are forced to zero.

3.5.6 Load Real Address (LRA)

Assembler Notation	Op-Code	Format
LRA R1,D2(X2)	63	RX1,RX2
LRA R1,A2(FX2,SX2)	63	RX3

Operation

This instruction simulates the operation of the Memory Address Translator. The register specified by R1 contains a program address (not relocated). The second operand address points to a relocation/protection module parameter block, in the format shown:

BYTE OFFSET	0	1	14	15	31	
+0		(PST ENTRIES) -1		A(PROCESS SEGMENT TABLE)/128		PSTD
+4		(SST ENTRIES) -1		A(SHARED SEGMENT TABLE)/128		SSTD

The address contained in the register specified by R1 is relocated, using the appropriate parameters. The relocated address replaces the contents of the register specified by R1.

Condition Code

C	V	G	L	
1	0	0	0	Segment not mapped
0	1	0	0	Nonpresent segment
0	0	1	X	Write-protected segment
0	0	X	1	Read- or Execute-protected segment
0	0	0	0	No restrictions

The condition code is determined on a priority basis with segment table size exceeded checked first, nonpresent segment second, segment limit exceeded third, and all protect keys (as a group) last.

Programming Notes

Segment tables must conform tc the rules given in the section on Memory Management; otherwise, the results of the LRA instruction are undefined.

If the address is not mapped or not present, the register specified by R1 is unchanged.

Segment table size exceeded or segment limit exceeded results in condition code 1000 (unmapped).

The second operand location must be on a fullword boundary.

Example: LRA

This example performs an address translation in the same manner as the Memory Address Translator (MAT). The steps shown are not optimal, and do not reflect the actual operation of the MAT, which is a high-speed device capable of performing several of the steps simultaneously.

To set up for this example, register R1 contains X'053147', the program address to be translated. RELOCBLK is the address of a relocation/protection module parameter block. This block contains two fullwords. The first of these is the Process Segment Table Descriptor (PSTD), with the value X'000E06BF'. The second is the Shared Segment Table Descriptor (SSTD), with the value X'000C06C0'. Memory location X'035FA8' contains the Process Segment Table Entry (PSTE) to be used, with the value X'588A0028'. Memory location X'036028' contains the Shared Segment Table Entry (SSTE) to be used, with the value X'58126880'. The instruction proceeds as follows:

```
LRA    R1,RELOCBLK          TRANSLATE ADDRESS IN R1
```

1. The PSTD is fetched from RELOCBLK, and ANDed with X'FFFE0000' to extract the segment table size field. The result, X'000E0000', is shifted right 17 bit positions, yielding X'00000007'. This value is the number of entries in the Process Segment Table (PST), minus one. Therefore, the PST has entries for segments 0 through 7.

2. The program address from register R1, X'053147', is shifted right 16 bit positions to yield the specified segment number, X'00000005'. The segment number is compared with the PST size. If the PST size were less than the segment number, this would mean that no entry existed in the PST for the specified segment, and that the segment was unmapped (condition code = 8). However, such is not the case, and the instruction proceeds.

3. The PSTD is ANDed with X'0001FFFF' to extract the segment table address field. The result, X'000006BF', is shifted left seven bit positions, to multiply it by 128. This yields the address of the PST, X'35F80'.

4. The segment number specified by the program address in R1 (X'053147') is used as an index into the PST. Because each Segment Table Entry (STE) requires eight bytes, the segment number, X'00000005', is shifted left three bit positions, to multiply it by eight. The result, X'00000028', and the address of the PST, X'035F80', are added. The result is the address X'035FA8', and the PSTE at that address is fetched. This PSTE has the value X'588A0028'.

The PSTE is ANDed with the value X'40000000' to test the Presence bit in the STE. If the bit were zero, this would mean the segment was not present (condition code = 4). But such is not the case, and the instruction proceeds.

5. The PSTE is then ANDed with X'00800000', to test the Shared Segment bit. If the bit were zero, the LRA instruction would use the data in the PSTE as data in the SSTE also, and perform the operations in step 9 below; but such is not the case.

   The Shared Segment bit in the PSTE is set, which means that an entry from the Shared Segment Table (SST) must also be used in translating the program address. The SSTD (X'000C06C0') is ANDed with X'FFFE0000' to extract the segment table size field. The result, X'000C0000', is shifted right 14 bit positions to yield X'00000030'. This value is the maximum SST offset, the offset in bytes from the start of the SST to the beginning of the last entry.

6. The SSTD is ANDed with X'0001FFFF' to extract the segment table address field. The result, X'000006C0', is shifted left seven bit positions to yield the address of the Shared Segment Table (SST), X'036000'.

7. The PSTE is now ANDed with X'0001FFFF' to extract the Segment Relocation Field (SRF). This field has the value X'00000028'. If this value exceeded the maximum SST offset, this would mean that no entry existed in the SST for the specified segment, and that the segment was unmapped (condition code = 8); but such is not the case, and the instruction proceeds. The SRF is added with the PST address, X'036000'. The Shared Segment Table Entry (SSTE) pointed to by the PSTE is located at the resulting address, X'036028'.

8. The SSTE is fetched, and its value found to be X'58126880'. This value is ANDed with X'40000000' to test the STE Presence bit. If the bit were zero, this would mean the segment was not present (condition code = 4); but such is not the case, and the instruction proceeds.

9. The SSTE, with a value X'58126880', is ANDed with the value X'003E0000' to extract the Segment Limit Field (SLF). The resulting value, X'00120000', is shifted right six bit positions, yielding an SLF value of X'00004800'. The program address from R1, X'053147', is ANDed with X'0000F800'. The resulting value, X'00003000', is compared to the SLF value, X'00004800'. If the SLF value were the lesser of the two values, this would indicate that the program address was in an unreachable part of the segment (segment limit violation), and thus unmapped (condition code = 8); but such is not the case, and the instruction proceeds.

10. At this point, address translation can be performed. The SSTE, with value X'58126880', is ANDed with the value X'0001FFFF' to extract the SRF. This field has the value X'00006880'. The SRF is shifted left seven bit positions, giving the relocation value X'00344400'.

    The program address from R1, X'053147', is ANDed with the value X'0000FFFF', giving the value X'00003147'. To this value is added the relocation value, X'00344000'. The result is the translated program address, X'347147', which replaces the contents of register R1.

11. The PSTE, with value X'588A0028', and the SSTE, with value X'58126880', are ANDed, yielding the value X'58020000'. This value contains the combined segment access keys. If ANDing the keys with X'08000000' yielded a zero result, the G flag would be set in the condition code to indicate a write-protected segment. If ANDing the keys with X'10000000' yielded a zero result, the L flag would be set in the condition code to indicate a read-protected segment; but neither is the case. ANDing the keys with X'04000000' does yield a zero result, and the L flag is set in the condition code to indicate that the segment is execute-protected. The LRA instruction terminates once these tests have been performed. (See Figure 3-5.)

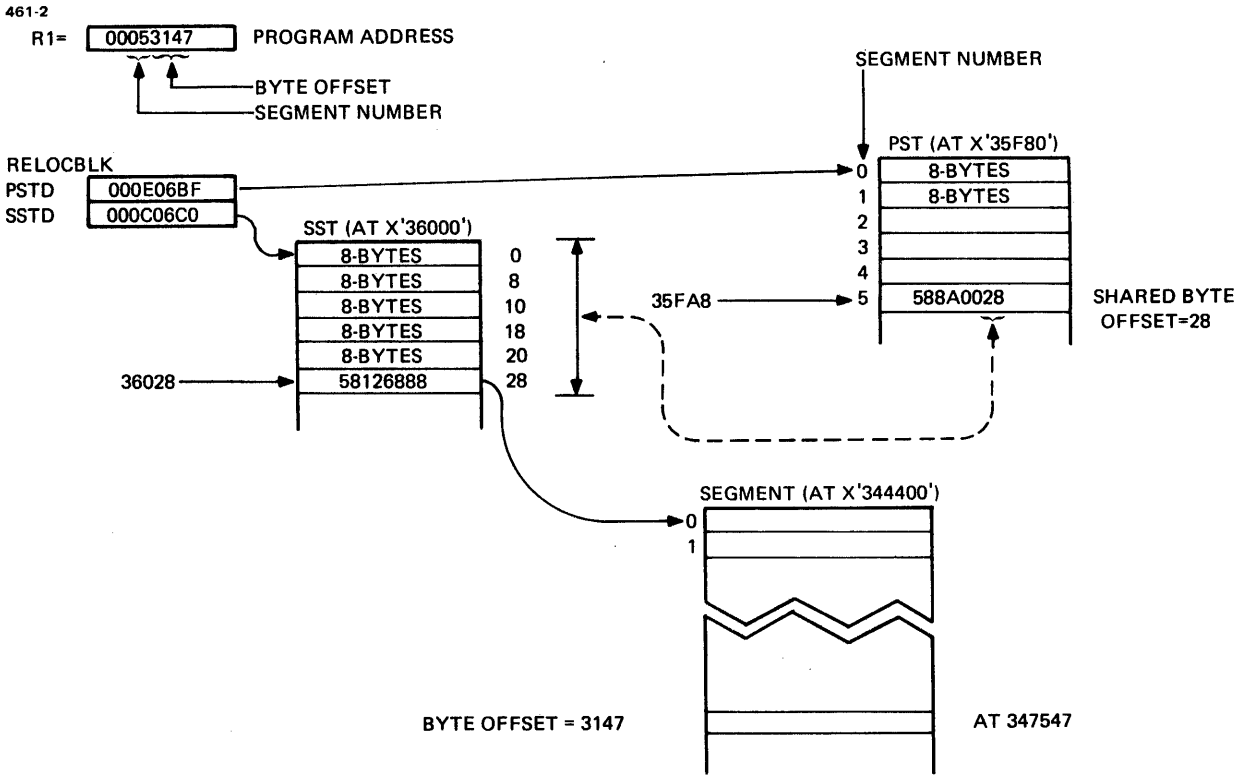


Figure 3-5 LRA Example

3.5.7 Load Halfword Logical (LHL)

Assembler Notation	Op-Code	Format
LHL R1,D2(X2)	73	RX1,RX2
LHL R1,A2(FX2,SX2)	73	RX3

Operation

The halfword second operand replaces bits 16:31 of the register specified by R1. Bits 0:15 of the register specified by R1 are replaced by zero.

Condition Code

C	V	G	L	
0	0	0	0	Value is zero
0	0	1	0	Value is not zero

Programming Note

The second operand must be located on a halfword boundary.

3.5.8 Load Multiple (LM)

Assembler Notation	Op-Code	Format
LM R1,D2(X2)	D1	RX1,RX2
LM R1,A2(FX2,SX2)	D1	RX3

Operation

Successive registers, starting with the register specified by R1, are loaded from successive memory locations, starting with the location specified as the effective address of the second operand. Each register is loaded with a fullword from memory. The process stops when register 15 has been loaded.

Condition Code

Unchanged

Programming Notes

The second operand must be located on a fullword boundary.

The second operand address is formed before any registers are loaded; therefore, X2, FX2, and SX2 can be among the registers loaded.

In the event of a machine malfunction due to a noncorrectable memory error, or due to a MAT fault, the effective address calculated at the beginning of the instruction is available, should a retry be desired. For details, refer to Chapter 10 and Chapter 12.

3.5.9 Load Byte (LB, LBR)

Load Byte (LB)
Load Byte Register (LBR)

Assembler Notation	Op-Code	Format
LB R1,D2(X2)	D3	RX1,RX2
LB R1,A2(FX2,SX2)	D3	RX3
LBR R1,R2	93	RR

Operation

The 8-bit second operand replaces the least significant bits (bits 24:31) of the register specified by R1. Bits 0:23 of the register are forced to zero.

Condition Code

Unchanged

Programming Note

In the Load Byte Register instruction, the second operand is taken from the least significant eight bits (bits 24:31) of the register specified by R2.

3.5.10 Exchange Halfword Register (EXHR)

Assembler Notation	Op-Code	Format
EXHR R1,R2	34	RR

Operation

Bits 0:15 of the register specified by R2 replace bits 16:31 of the register specified by R1. Bits 16:31 of the register specified by R2 replace bits 0:15 of the register specified by R1.

Condition Code

Unchanged

Programming Note

If R1 equals R2, the two halfwords contained within the register are exchanged. If R1 does not equal R2, the contents of R2 are unchanged.

Example: EXHR

Assembler Notation		Machine Code	Comments
LI	REG5,Y'0ABCDEF9'	F850 0ABC DEF9	(REG5) = 0ABCDEF9
LI	REG7,Y'12345678'	F870 1234 5678	(REG7) = 12345678
EXHR	REG5,REG7	3457	

Result of EXHR Instruction

(REG5) = 56781234
(REG7) = 12345678
Condition Code unchanged

3.5.11 Exchange Byte Register (EXBR)

Assembler Notation	Op-Code	Format
EXBR R1,R2	94	RR

Operation

The two 8-bit bytes contained in bits 16:31 of the register specified by R2 are exchanged and loaded into bits 16:31 of the register specified by R1. Bits 0:15 of the register specified by R1 are unchanged. The register specified by R2 is unchanged.

Condition Code

Unchanged

Programming Note

R1 and R2 may specify the same register. In this case, the two bytes in bits 16:31 of the register specified by R2 are exchanged.

Example: EXBR

Assembler Notation		Machine Code	Comments
LI	REG7,X'5A6B3C4D'	F870 5A6B 3C4D	(REG7) = 5A6B3C4D
LI	REG3,Y'98761234'	F830 9876 1234	(REG3) = 98761234
EXBR	REG7,REG3	9473	

Result of EXBR Instruction

(REG7) = 5A6B3412
(REG3) = 98761234
Condition Code unchanged

3.5.12 Store (ST)

Assembler Notation	Op-Code	Format
ST R1,D2(X2)	50	RX1,RX2
ST R1,A2(FX2,SX2)	50	RX3

Operation

The 32-bit contents of the register specified by R1 replace the contents of the fullword memory location specified by the effective address of the second operand.

Condition Code

Unchanged

Programming Note

The second operand location must be on a fullword boundary.

3.5.13 Store Halfword (STH)

Assembler Notation	Op-Code	Format
STH R1,D2(X2)	40	RX1,RX2
STH R1,A2(FX2,SX2)	40	RX3

Operation

Bits 16:31 of the register specified by R1 replace the contents of the halfword memory location specified by the effective address of the second operand.

Condition Code

Unchanged

Programming Note

The second operand location must be on a halfword boundary.

3.5.14 Store Multiple (STM)

Assembler Notation	Op-Code	Format
STM R1,D2(X2)	D0	RX1,RX2
STM R1,A2(FX2,SX2)	D0	RX3

Operation

The fullword contents of registers, starting with the register specified by R1, replace the contents of successive fullword memory locations, starting with the location specified by the effective address of the second operand. The process stops when register 15 has been stored.

Condition Code

Unchanged

Programming Note

The second operand location must be on a fullword boundary.

3.5.15 Store Byte (STB, STBR)

Store Byte (STB)
Store Byte Register (STBR)

Assembler Notation		Op-Code	Format
STB	R1,D2(X2)	D2	RX1,RX2
STB	R1,A2(FX2,SX2)	D2	RX3
STBR	R1,R2	92	RR

Operation

The least significant eight bits (bits 24:31) of the register specified by R1 are stored in the byte second operand location.

Condition Code

Unchanged

Programming Note

In the Store Byte Register instruction, the 8-bit quantity is stored in bits 24:31 of the register specified by R2. Bits 0:23 of the register are unchanged.

Example: STBR

Assembler Notation	Machine Code	Comments
LI REG4,Y'13577531'	F840 1357 7531	(REG4) = 13577531
LI REG3,Y'24688642'	F830 2468 8642	(REG3) = 24688642
.		
.		
.		
STBR REG4,REG3	9243	

Result of STBR Instruction

(REG4) = 13577531
(REG3) = 24688631
Condition Code unchanged

3.5.16 Compare Logical (CL, CLR, CLI)

Compare Logical (CL)
Compare Logical Register (CLR)
Compare Logical Immediate (CLI)

Assembler Notation		Op-Code	Format
CL	R1,D2(X2)	55	RX1,RX2
CL	R1,A2(FX2,SX2)	55	RX3
CLR	R1,R2	05	RR
CLI	R1,I2(X2)	F5	RI2

Operation

The first operand, the contents of the register specified by R1, is compared logically to the second operand. The result is indicated by the condition code setting. Neither operand is changed.

Condition Code

C	V	G	L	
0	X	0	0	First operand equal to second
1	X	0	1	First operand less than second
1	X	1	0	First operand less than second
0	X	0	1	First operand greater than second
0	X	1	0	First operand greater than second

Programming Notes

In the RX formats, the second operand must be located on a fullword boundary.

The state of the V flag is undefined.

If the second operand is zero, the C flag cannot set.

It is meaningful to check the following condition code mask (M1) after a logical comparison:

Mask	True/False*	Inference
3	False	First operand equal to second
3	True	First operand not equal to second
8	False	First operand greater than or equal to second
8	True	First operand less than second

*Refer to Chapter 4 for True/False concept in branch instructions.

3.5.17 Compare Logical Halfword (CLH, CLHI)

Compare Logical Halfword (CLH)
Compare Logical Halfword Immediate (CLHI)

Assembler Notation		Op-Code	Format
CLH	R1,D2(X2)	45	RX1,RX2
CLH	R1,A2(FX2,SX2)	45	RX3
CLHI	R1,I2(X2)	C5	RI1

Operation

The halfword second operand is expanded to a fullword by propagating the most significant bit through bits 0:15. The first operand, the contents of the register specified by R1, is compared to this fullword. The result is indicated by the condition code setting. Neither operand is changed.

Condition Code

C	V	G	L	
0	X	0	0	First operand equal to second
1	X	0	1	First operand less than second
1	X	1	0	First operand less than second
0	X	0	1	First operand greater than second
0	X	1	0	First operand greater than second

Programming Notes

In the RX formats, the second operand must be located on a halfword boundary.

In the RI1 format, the 16-bit I2 field is extended to a fullword by propagating the sign bit through bits 0:15. The contents of the index register specified by X2 are then added to form the fullword second operand.

The state of the V flag is undefined.

If the second operand is zero, the C flag cannot set.

It is meaningful to check the following condition code mask (M1) after a logical comparison:

Mask	True/False*	Inference
3	False	First operand equal to second
3	True	First operand not equal to second
8	False	First operand greater than or equal to second
8	True	First operand less than second

*Refer to Chapter 4 for True/False concept in branch instructions.

3.5.18 Compare Logical Byte (CLB)

Assembler Notation	Op-Code	Format
CLB R1,D2(X2)	D4	RX1,RX2
CLB R1,A2(FX2,SX2)	D4	RX3

Operation

The byte quantity, contained in bits 24:31 of the register specified by R1, is compared with the 8-bit second operand. The result is indicated by the condition code setting. Neither operand is changed.

Condition Code

C	V	G	L	
0	X	0	0	First operand equal to second
1	X	0	1	First operand less than second
0	X	1	0	First operand greater than second

Programming Notes

Both operands are treated as unsigned quantities.

If the second operand is zero, the C flag cannot set.

It is meaningful to check the following condition code mask (M1) after a logical comparison:

Mask	True/False*	Inference
2	False	First operand not greater than second
2	True	First operand greater than second
3	False	First operand equal to second
3	True	First operand not equal to second
8	False	First operand greater than or equal to second
8	True	First operand less than second

*Refer to Chapter 4 for True/False concept in branch instructions.

3.5.19 AND (N, NR, NI)

AND (N)
AND Register (NR)
AND Immediate (NI)

Assembler Notation		Op-Code	Format
N	R1,D2(X2)	54	RX1,RX2
N	R1,A2(FX2,SX2)	54	RX3
NR	R1,R2	04	RR
NI	R1,I2(X2)	F4	RI2

Operation

The logical product of the 32-bit second operand and the contents of the register specified by R1 replace the contents of the register specified by R1. The 32-bit logical product is formed on a bit-by-bit basis.

Condition Code

C	V	G	L	
0	0	0	0	Result is zero
0	0	0	1	Result is not zero
0	0	1	0	Result is not zero

Programming Notes

In the RX formats, the second operand must be located on a fullword boundary.

When operating on fixed-point data, the condition code indicates zero (no flags), negative (L flag), or positive (G flag) result.

3.5.20 AND Halfword (NH, NHI)

AND Halfword (NH)
AND Halfword Immediate (NHI)

Assembler Notation		Op-Code	Format
NH	R1,D2(X2)	44	RX1,RX2
NH	R1,A2(FX2,SX2)	44	RX3
NHI	R1,I2(X2)	C4	RI1

Operation

The halfword second operand is expanded to a fullword by propagating the most significant bit through bits 0:15. The logical product of this 32-bit quantity and the contents of the register specified by R1 replace the contents of the register specified by R1. The 32-bit logical product is formed on a bit-by-bit basis.

Condition Code

C	V	G	L	
0	0	0	0	Result is zero
0	0	0	1	Result is not zero
0	0	1	0	Result is not zero

Programming Notes

In the RX formats, the second operand must be located on a halfword boundary.

In the RI1 format, the 16-bit I2 field is extended to a fullword by propagating the sign bit through bits 0:15. The contents of the index register specified by X2 are then added to form the fullword second operand.

When operating on fixed-point data, the condition code indicates zero (no flags), negative (L flag), or positive (G flag) result.

3.5.21 OR (O, OR, OI)

OR (O)
OR Register (OR)
OR Immediate (OI)

Assembler Notation		Op-Code	Format
O	R1,D2(X2)	56	RX1,RX2
O	R1,A2(FX2,SX2)	56	RX3
OR	R1,R2	06	RR
OI	R1,I2(X2)	F6	RI2

Operation

The logical sum of the 32-bit second operand and the contents of the register specified by R1 replace the contents of the register specified by R1. The 32-bit logical sum is formed on a bit-by-bit basis.

Condition Code

C	V	G	L	
0	0	0	0	Result is zero
0	0	0	1	Result is not zero
0	0	1	0	Result is not zero

Programming Notes

In the RX formats, the second operand must be located on a fullword boundary.

When operating on fixed-point data, the condition code indicates zero (no flags), negative (L flag), or positive (G flag) result.

3.5.22 OR Halfword (OH, OHI)

OR Halfword (OH)
OR Halfword Immediate (OHI)

Assembler Notation		Op-Code	Format
OH	R1,D2(X2)	46	RX1,RX2
OH	R1,A2(FX2,SX2)	46	RX3
OHI	R1,I2(X2)	C6	RI1

Operation

The halfword second operand is expanded to a fullword by propagating the most significant bit through bits 0:15. The logical sum of this 32-bit quantity and the contents of the register specified by R1 replace the contents of the register specified by R1. The 32-bit logical sum is formed on a bit-by-bit basis.

Condition Code

C	V	G	L	
0	0	0	0	Result is zero
0	0	0	1	Result is not zero
0	0	1	0	Result is not zero

Programming Notes

In the RX formats, the second operand must be located on a halfword boundary.

In the RI1 format, the 16-bit I2 field is extended to a fullword by propagating the sign bit through bits 0:15. The contents of the index register specified by X2 are then added to form the fullword second operand.

When operating on fixed-point data, the condition code indicates zero (no flags), negative (L flag), or positive (G flag) result.

3.5.23 Exclusive OR (X, XR, XI)

Exclusive OR (X)
Exclusive OR Register (XR)
Exclusive OR Immediate (XI)

Assembler Notation		Op-Code	Format
X	R1,D2(X2)	57	RX1,RX2
X	R1,A2(FX2,SX2)	57	RX3
XR	R1,R2	07	RR
XI	R1,I2(X2)	F7	RI2

Operation

The logical difference of the 32-bit second operand and the contents of the register specified by R1 replace the contents of the register specified by R1. The 32-bit logical difference is formed on a bit-by-bit basis.

Condition Code

C	V	G	L	
0	0	0	0	Result is zero
0	0	0	1	Result is not zero
0	0	1	0	Result is not zero

Programming Notes

In the RX formats, the second operand must be located on a fullword boundary.

When operating on fixed-point data, the condition code indicates zero (no flags), negative (L flag), or positive (G flag) result.

3.5.24 Exclusive OR Halfword (XH, XHI)

Exclusive OR Halfword (XH)
Exclusive OR Halfword Immediate (XHI)

Assembler Notation		Op-Code	Format
XH	R1,D2(X2)	47	RX1,RX2
XH	R1,A2(FX2,SX2)	47	RX3
XHI	R1,I2(X2)	C7	RI1

Operation

The halfword second operand is expanded to a fullword by propagating the most significant bit through bits 0:15. The logical difference of this 32-bit quantity and the contents of the register specified by R1 replace the contents of the register specified by R1. The 32-bit logical difference is formed on a bit-by-bit basis.

Condition Code

C	V	G	L	
0	0	0	0	Result is zero
0	0	0	1	Result is not zero
0	0	1	0	Result is not zero

Programming Notes

In the RX formats, the second operand must be located on a halfword boundary.

In the RI1 format, the 16-bit I2 field is extended to a fullword by propagating the sign bit through bits 0:15. The contents of the index register specified by X2 are then added to form the fullword second operand.

When operating on fixed-point data, the condition code indicates zero (no flags), negative (L flag), or positive (G flag) result.

3.5.25 Test Immediate (TI)

Assembler Notation	Op-Code	Format
TI R1,I2(X2)	F3	RI2

Operation

Each bit of the second operand is logically ANDed with the corresponding bit in the register specified by R1. Neither operand is changed.

Condition Code

C	V	G	L	
0	0	0	0	Result is zero
0	0	0	1	Result is not zero
0	0	1	0	Result is not zero

Programming Notes

When operating on fixed-point data, the condition code indicates zero (no flags), negative (L flag), or positive (G flag) result.

This instruction works the same as the AND Immediate instruction (NI) except that the first operand is not changed.

Example: TI

This example tests if bit 16 of register 9 is set.

(REG9) = 7EFBC230

Assembler Notation		Comments
TI	REG9,Y'00008000'	Test bit 16
BNZ	LABEL	Branch if bit is set

Result of TI Instruction

(REG9) Unchanged
Condition Code = 0010 (G=1)
The conditional branch is taken.

3.5.26 Test Halfword Immediate (THI)

Assembler Notation	Op-Code	Format
THI R1,I2(X2)	C3	RI1

Operation

The halfword second operand is expanded to a fullword by propagating the most significant bit through bits 0:15. Each bit in this quantity is logically ANDed with the corresponding bit contained in the register specified by R1. Neither operand is changed.

Condition Code

C	V	G	L	
0	0	0	0	Result is zero
0	0	0	1	Result is not zero
0	0	1	0	Result is not zero

Programming Notes

When operating on fixed-point data, the condition code indicates zero (no flags), negative (L flag), or positive (G flag) result.

In the RI1 format, the 16-bit I2 field is extended to a fullword by propagating the sign bit through bits 0:15. The contents of the index register specified by X2 are then added to form the fullword second operand.

This instruction works the same as the AND Halfword Immediate instruction (NHI) except that the first operand is not changed.

Example: THI

This example tests if any of bits 0:16 of register 9 is set.

(REG9) = 80800000

Assembler Notation		Comments
THI	REG9,X'8000'	Test bits 0:16
BNZ	LABEL	Branch if any set

Result of THI Instruction

(REG9) Unchanged
Condition Code = 0001 (L=1)
The conditional branch is taken.

3.5.27 Shift Left Logical (SLL, SLLS)

Shift Left Logical (SLL)
Shift Left Logical Short (SLLS)

Assembler Notation		Op-Code	Format
SLL	R1,I2(X2)	ED	RI1
SLLS	R1,N	11	SF

Operation

The first operand, the contents of the register specified by R1, is shifted left the number of places specified by the second operand. Bits shifted out of position 0 are shifted through the carry flag of the condition code and then lost. The last bit shifted remains in the carry flag. Zeros are shifted into position 31.

Condition Code

C	V	G	L	
X	0	0	0	Result is zero
X	0	0	1	Result is not zero
X	0	1	0	Result is not zero
1	0	X	X	Carry

Programming Notes

In the RI1 format, the shift count is specified by the least significant five bits of the second operand. The maximum shift count is 31.

In the SF format, the maximum shift count is 15.

The state of the C flag indicates the state of the last bit shifted out of position 0.

If the second operand specifies a shift of zero places, the condition code is set in accordance with the value contained in the register. The C flag is zero in this case.

When the register specified by R1 contains fixed-point data, the L flag set indicates a negative result; the G flag set indicates a positive result.

3.5.28 Shift Right Logical (SRL, SRLS)

Shift Right Logical (SRL)
Shift Right Logical Short (SRLS)

Assembler Notation		Op-Code	Format
SRL	R1,I2(X2)	EC	RI1
SRLS	R1,N	10	SF

Operation

The first operand, the contents of the register specified by R1, is shifted right the number of places specified by the second operand. Bits shifted out of position 31 are shifted through the carry flag of the condition code and then lost. The last bit shifted remains in the carry flag. Zeros are shifted into position 0.

Condition Code

C	V	G	L	
X	0	0	0	Result is zero
X	0	0	1	Result is not zero
X	0	1	0	Result is not zero
1	0	X	X	Carry

Programming Notes

In the RI1 format, the shift count is specified by the least significant five bits of the second operand. The maximum shift count is 31.

In the SF format, the maximum shift count is 15.

The state of the C flag indicates the state of the last bit shifted out of position 31.

When the register specified by R1 contains fixed-point data, the L flag set indicates a negative result; the G flag set indicates a positive result.

If the second operand specifies a shift of zero places, the condition code is set in accordance with the value contained in the register. The C flag is zero in this case.

3.5.29 Shift Left Halfword Logical (SLHL, SLHLS)

Shift Left Halfword Logical (SLHL)
Shift Left Halfword Logical Short (SLHLS)

Assembler Notation		Op-Code	Format
SLHL	R1,I2(X2)	CD	RI1
SLHLS	R1,N	91	SF

Operation

Bits 16:31 of the register specified by R1 are shifted left the number of places specified by the second operand. Bits shifted out of position 16 are shifted through the carry flag and lost. The last bit shifted remains in the carry flag. Zeros are shifted into position 31. Bits 0:15 of the first operand remain unchanged.

Condition Code

C	V	G	L	
X	0	0	0	Result is zero
X	0	0	1	Result is not zero
X	0	1	0	Result is not zero
1	0	X	X	Carry

Programming Notes

The condition code setting is based on the halfword (bits 16:31) result.

In the RI1 format, the shift count is specified by the least significant four bits of the second operand. The maximum shift count is 15.

In the SF format, the maximum shift count is 15.

The state of the C flag indicates the state of the last bit shifted out of position 16.

When the register specified by R1 contains fixed-point data, the L flag set indicates a negative result; the G flag set indicates a positive result.

If the second operand specifies a shift of zero places, the condition code is set in accordance with the value contained in bits 16:31 of the register. The C flag is zero in this case.

3.5.30 Shift Right Halfword Logical (SRHL, SRHLS)

Shift Right Halfword Logical (SRHL)
Shift Right Halfword Logical Short (SRHLS)

Assembler Notation		Op-Code	Format
SRHL	R1,I2(X2)	CC	RI1
SRHLS	R1,N	90	SF

Operation

Bits 16:31 of the register specified by R1 are shifted right the number of places specified by the second operand. Bits shifted out of position 31 are shifted through the carry flag and lost. The last bit shifted remains in the carry flag. Zeros are shifted into position 16. Bits 0:15 of the first operand remain unchanged.

Condition Code

C	V	G	L	
X	0	0	0	Result is zero
X	0	0	1	Result is not zero
X	0	1	0	Result is not zero
1	0	X	X	Carry

Programming Notes

The condition code setting is based on the halfword (bits 16:31) result.

In the RI1 format, the shift count is specified by the least significant four bits of the second operand. The maximum shift count is 15.

In the SF format, the maximum shift count is 15.

The state of the C flag indicates the state of the last bit shifted out of position 31.

When the register specified by R1 contains fixed-point data, the L flag set indicates a negative result; the G flag set indicates a positive result.

If the second operand specifies a shift of zero places, the condition code is set in accordance with the halfword value contained in bits 16:31 of the register. The C flag is zero in this case.

3.5.31 Rotate Left Logical (RLL)

Assembler Notation	Op-Code	Format
RLL R1,I2(X2)	EB	RI1

Operation

The 32-bit first operand, contained in the register specified by R1, is shifted left, end around, the number of positions specified by the second operand. Bits shifted out of position 0 are shifted into position 31.

Condition Code

C	V	G	L	
0	0	0	0	Result is zero
0	0	0	1	Result is not zero
0	0	1	0	Result is not zero

Programming Notes

The shift count is specified by the least significant five bits of the second operand. The maximum shift count is 31.

When the register specified by R1 contains fixed-point data, the L flag set indicates a negative result; the G flag set indicates a positive result.

If the second operand specifies a shift of zero places, the condition code is set in accordance with the value contained in the register specified by R1.

Example: RLL

1.

Assembler Notation	Machine Code	Comments
LI REG9,Y'56789ABC'	F890 56789ABC	(REG9)=56789ABC
RLL REG9,X'0004'	EB90 0004	

Result of RLL Instruction

(REG9) = 6789ABC5
Condition Code = 0010 (G=1)

2.

Assembler Notation	Machine Code	Comments
LI REG9,Y'88880000'	F890 8888 0000	(REG9)=88880000
RLL REG9,X'03'	EB90 0003	

Result of RLL Instruction

(REG9) = 44400004
Condition Code = 0010 (G=1)

3.5.32 Rotate Right Logical (RRL)

Assembler Notation	Op-Code	Format
RRL R1,I2(X2)	EA	RI1

Operation

The 32-bit first operand, contained in the register specified by R1, is shifted right, end around, the number of positions specified by the second operand. Bits shifted out of position 31 are shifted into position 0.

Condition Code

C	V	G	L	
0	0	0	0	Result is zero
0	0	0	1	Result is not zero
0	0	1	0	Result is not zero

Programming Notes

The shift count is specified by the least significant five bits of the second operand. The maximum shift count is 31.

When the register specified by R1 contains fixed-point data, the L flag set indicates a negative result; the G flag set indicates a positive result.

If the second operand specifies a shift of zero places, the condition code is set in accordance with the value contained in the register specified by R1.

Example: RRL

1.

Assembler Notation		Machine Code	Comments
LI	REG4,Y'12345678'	F840 1234 5678	(REG4) = 12345678
RRL	REG4,X'04'	EA40 0004	

Result of RRL Instruction

(REG4) = 81234567
Condition Code = 0001 (L=1)

2.

Assembler Notation		Machine Code	Comments
LI	REG4,Y'00001111'	F840 0000 1111	(REG4) = 00001111
RRL	REG4,X'01'	EA40 0001	

Result of RRL Operation

(REG4) = '800000888'
Condition Code = 0001 (L=1)

3.5.33 Test and Set (TS)

Assembler Notation	Op-Code	Format
TS D2(X2)	E0	RX1,RX2
TS A2(FX2,SX2)	E0	RX3

Operation

The halfword operand is read from memory and, on the same cycle, written back with the most significant bit set. The other bits in the halfword are unchanged. On the read cycle, the most significant bit of the operand is tested. The condition code reflects the state of this bit at the time of the memory read.

Condition Code

C	V	G	L	
X	X	X	0	Most significant bit is zero
X	X	X	1	Most significant bit is set

Programming Notes

The second operand must be located on a halfword boundary.

The TS instruction provides a mechanism for software synchronization and can be used in a single processor environment as follows: Two or more user tasks running under an operating system share a halfword. This halfword is located in a memory area referred to as Task Common. Each task can access the halfword using the TS instruction. The synchronization sequence may be as follows:

TASK 1 Sets the most significant bit using the TS instruction.

TASK 2 Senses the most significant bit using the TS instruction, sees that it is set, performs the necessary software synchronization, and then zeros the most significant bit of the halfword.

The TS instruction can be used in a multiprocessor system as follows: two or more processors share a halfword. This halfword is located in a memory area referred to as Shared Memory. Each processor can access the halfword using the TS instruction. The synchronization sequence can be as explained for user tasks with the following slight difference: whereas TASK 1 and TASK 2 cannot access the halfword at the same (real) time, two processors can. The access is granted according to the relative priority of the two processors.

The hardware ensures that no other accesses to the halfword are made during the execution of the TS instruction.

3.5.34 Test Bit (TBT)

Assembler Notation		Op-Code	Format
TBT	R1,D2(X2)	74	RX1,RX2
TBT	R1,A2(FX2,SX2)	74	RX3

Operation

The second operand address points to a bit array starting on a byte boundary. The value contained in the register specified by R1 is the bit displacement into the array. Bits in the array are counted from left to right starting with bit 0. The argument bit is located and tested. The test does not change the bit.

Condition Code

C	V	G	L	
0	0	0	0	Tested bit is zero
0	0	1	0	Tested bit is one

Programming Note

For software compatibility with other processors, the bit array should start on a halfword boundary.

Example: TBT

Assembler Notation		Machine Code	Comments
LIS	REG8,3	2483	(REG8) = 3
TBT	REG8,LABEL	7480 0BC4	LABEL = halfword in memory at location X'0BC4'. It contains X'B34A'.

Result of TBT Instruction

Memory Location X'BC4' unchanged
(REG8) unchanged
Condition Code = 0010 (G=1)...Bit 3 of location X'BC4' is set.

3.5.35 Set Bit (SBT)

Assembler Notation		Op-Code	Format
SBT	R1,D2(X2)	75	RX1,RX2
SBT	R1,A2(FX2,SX2)	75	RX3

Operation

The second operand address points to a bit array starting on a byte boundary. The value contained in the register specified by R1 is the bit displacement into the array. Bits in the array are counted from left to right starting with bit 0. The argument bit is located and set to one.

Condition Code

C	V	G	L	
0	0	0	0	Previous state of bit was zero
0	0	1	0	Previous state of bit was one

Programming Note

For software compatibility with other processors, the bit array should start on a halfword boundary.

Example: SBT

Assembler Notation		Machine Code	Comments
LIS	REG5,8	2458	(REG5) = 8
SBT	REG5,LABEL	7550 1520	LABEL located at X'1520'. It contains X'2134'.

Result of SBT Instruction

Contents of LABEL = 21B4
(REG5) unchanged
Condition Code = 0000 (G=0)

3.5.36 Reset Bit (RBT)

Assembler Notation		Op-Code	Format
RBT	R1,D2(X2)	76	RX1,RX2
RBT	R1,A2(FX2,SX2)	76	RX3

Operation

The second operand address points to a bit array starting on a byte boundary. The value contained in the register specified by R1 is the bit displacement into the array. Bits in the array are counted from left to right starting with bit zero. The argument bit is located and forced to zero (reset).

Condition Code

C	V	G	L	
0	0	0	0	Previcus state of bit was zero
0	0	1	0	Previcus state of bit was one

Programming Note

For software compatibility with other processors, the bit array should start on a halfword boundary.

Example: RBT

Assembler Notation		Machine Code	Comments
LIS	REG2,3	2423	(REG2) = 3
RBT	REG2,LABEL	7620 1A42	LABEL located at X'1A42' contains X'3143'.

Result of RBT Instruction

Contents of LABEL = 2143
(REG2) unchanged
Condition Code = 0010 (G=1)

3.5.37 Complement Bit (CBT)

Assembler Notation		Op-Code	Format
CBT	R1,D2(X2)	77	RX1,RX2
CBT	R1,A2(FX2,SX2)	77	RX3

Operation

The second operand address points to a bit array starting on a byte boundary. The value contained in the register specified by R1 is the bit displacement into the array. Bits in the array are counted from left to right starting with bit 0. The argument bit is located and complemented.

Condition Code

C	V	G	L	
0	0	0	0	Previous state of bit was zero
0	0	1	0	Previcus state of bit was one

Programming Note

For software compatibility with other processors, the bit array should start on a halfword boundary.

Example: CBT

Assembler Notation		Machine Code	Comments
LIS	REG9,3	2493	(REG9) = 3
CBT	REG9,LABEL	7790 0C4A	LABEL located at X'C4A'. It contains X'2813'.

Result of CBT Instruction

Contents of LABEL = 3813
(REG9) unchanged
Condition Code = 0000 (G=0)

3.5.38 Cyclic Redundancy Check

Cyclic Redundancy Check Modulo 12 (CRC12)
Cyclic Redundancy Check Modulo 16 (CRC16)

Assembler Notation		Op-Code	Format
CRC12	R1,D2(X2)	5E	RX1,RX2
CRC12	R1,A2(FX2,SX2)	5E	RX3
CRC16	R1,D2(X2)	5F	RX1,RX2
CRC16	R1,A2(FX2,SX2)	5F	RX3

Operation

These instructions are used to generate either a 12-bit or a 16-bit Cyclic Redundancy Check (CRC) residual halfword. The register specified by R1 contains, in bits 24:31, the data character to be included in the CRC residual. The second operand is the accumulated (old) CRC residual. The polynomial used for the 12-bit CRC generation is:

$$X^{12} + X^{11} + X^3 + X^2 + X + 1$$

The polynomial used for the 16-bit CRC generation is:

$$X^{16} + X^{15} + X^2 + 1$$

The halfword second operand is replaced by the generated CRC residual.

Condition Code

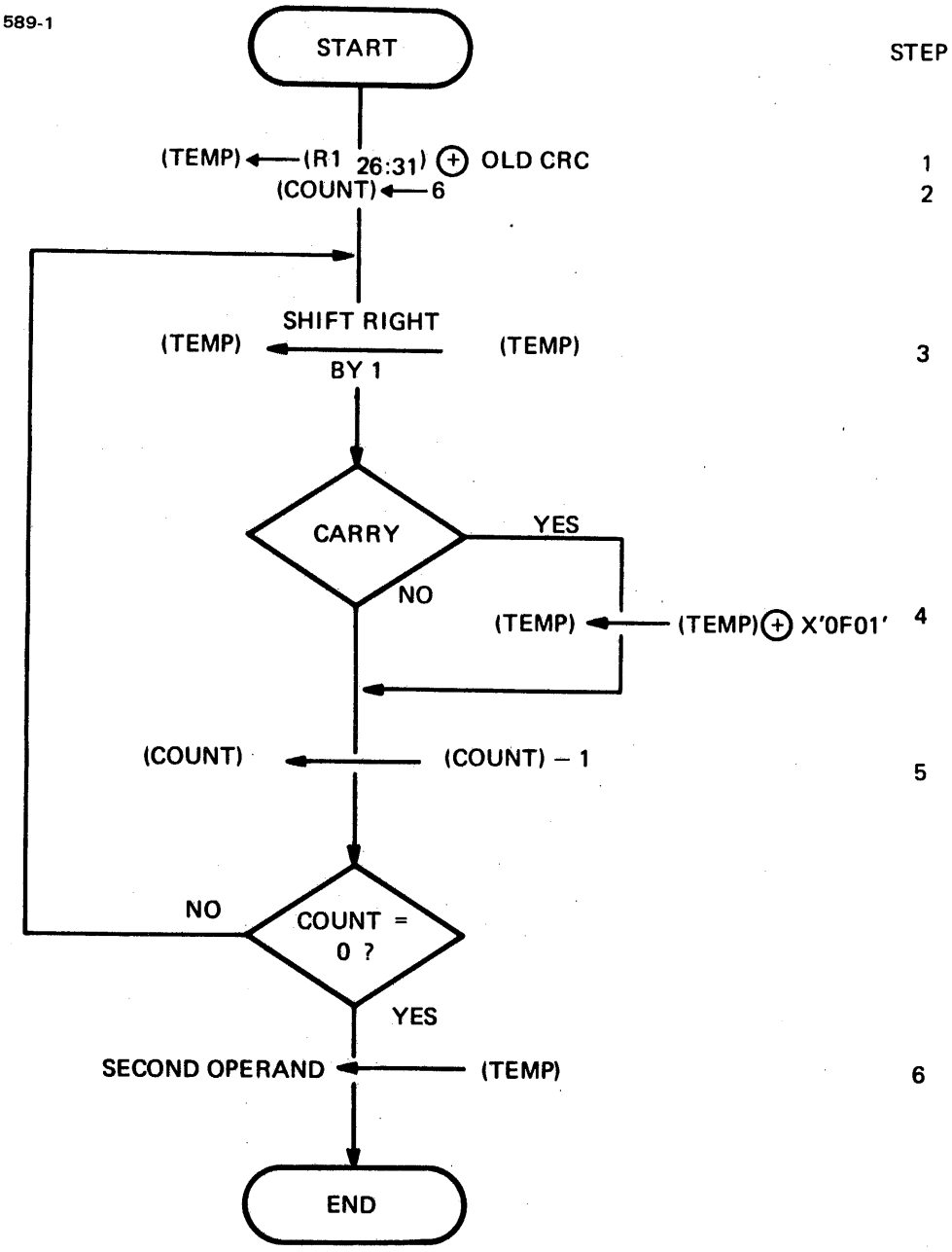
Unchanged

Programming Notes

The register specified by R1 remains unchanged.

The second operand must be located on a halfword boundary.

Figure 3-6 illustrates a flowchart for CRC generation.



CRC12 ALGORITHM SHOWN

FOR CRC 16 ALGORITHM, USE:	$R1_{24:31}$	INSTEAD OF $R1_{26:31}$	IN STEP 1
	8	INSTEAD OF 6	IN STEP 2
	X'A001'	INSTEAD OF X'0F01'	IN STEP 4

Figure 3-6 Flowchart for CRC Generation

3.5.39 Translate (TLATE)

Assembler Notation		Op-Code	Format
TLATE	R1,D2(X2)	E7	RX1,RX2
TLATE	R1,A2(FX2,SX2)	E7	RX3

Operation

The least significant eight bits (bits 24:31) of the register specified by R1 contain the character to be translated. The fullword location specified by the second operand address contains the address of a translation table. The table is made up of 256 halfwords. The character contained in the register specified by R1 is used as an index into the table.

If bit 0 of the table entry corresponding to the index character is one, bits 8:15 of the table entry replace the index character, and the next sequential instruction is executed.

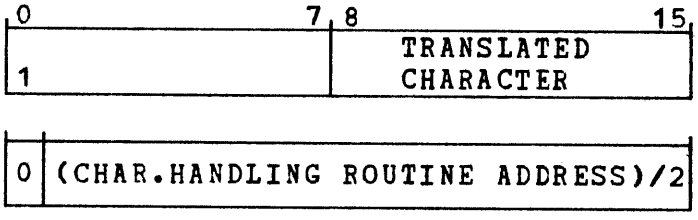
If bit 0 of the table entry is zero, bits 1:15 of the table entry contain the address, divided by two, of a special character handling routine. In this case, no translation takes place. The address contained in bits 1:15 is shifted left by one (multiplied by two). This address replaces the current location counter, thereby effecting an unconditional branch to the special character handling routine. Translation of character string data may also be performed using the MVTU instruction. (See Chapter 7.)

Condition Code

Unchanged

Programming Notes

The second operand address must be located on a fullword boundary.



Example: TLATE

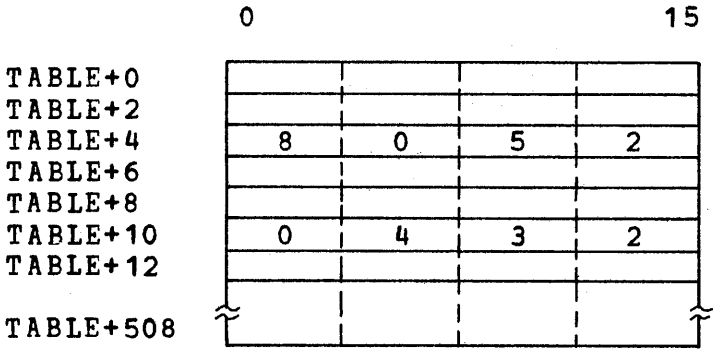
This example illustrates the use of the TLATE instruction. The translation table must either be initialized or assembled to contain up to a total of 256 halfword entries. In this example, the table contains 2 entries:

```
Label     Assembler Notation          Comments

          LHI    REG5, X'8052'    LOAD TABLE ENTRY INTO REG5
          STH    REG5,TABLE       PUT ENTRY INTO TABLE
          LA     REG7,TRANLAB     LOAD ANOTHER TABLE ENTRY
          SRLS   REG7,1           DIVIDE BY 2
          STH    REG7,TABLE+4     PUT ENTRY INTO TABLE
          .
          .
          .
TABADR    DC     A(TABLE)
```

Alternatively, this table may be assembled with the proper constant values. The T type constant may be used to assemble subroutine addresses in the proper format. For example:

```
      ALIGN   2
TABLE EQU     *
      DO      256
      DC      H'0'
      ORG     TABLE+4
      DC      T(TRANLAB)
      ORG     TABLE+512
```

Since a program is normally assembled as a relocatable program, the address of TRANLAB is not known, but for illustrative purposes assume the address of TRANLAB is X'864'.



At TABLE+10 is the address cf TRANLAB divided by 2 (X'864'/2)

1. Using this table, this example translates the character in register 2.

```
Label     Assembler Notation          Comments

          LIS      REG2,2          (REG2) = 0000 0002
          TLATE    REG2,TABADR
```

Result of TLATE Instruction

(REG2) = 0000 0052
Condition Code unchanged

The entry used = data at address of (2 times contents of REG2) + TABLE
= data at address TABLE + 4
= X'8052'

Since the first bit of the entry is 1, direct translation is used and the contents of REG2 are replaced by X'0000 0052'.

2. Using the table, the following example shows how the TLATE instruction can be used to branch to a special character handling routine:

```
Label      Assembler Notation          Comments

           LIS     REG5,5              (REG5) = 0000 0005
           TLATE   REG5,TABADR
           .
           .
           .
           .
           .
TRANLAB    LR   R6,R5                  THESE INSTRUCTIONS
           LB   R3,0(R6)               OPERATE ON THE
           .                           SPECIAL CHARACTER.
           .
           .

           .
           .
           .
```

Result of TLATE Instruction (continued)

(REG5) = 0000 0005
Condition Code unchanged

Control is transferred to the subroutine at address TRANLAB (X'864').

The entry used = data at address of (2 times contents of REG5) + TABLE
= data at address TABLE + A
= X'0432'

Since the first bit of the entry is 0, the entry is multiplied by 2, a transfer occurs to TRANLAB (at address X'864'), and the processor executes instructions from the new address.

3.5.40 Add To List (ATL, ABL)

Add to Top of List (ATL)
Add to Bottom of List (ABL)

Assembler Notation		Op-Code	Format
ATL	R1,D2(X2)	64	RX1,RX2
ATL	R1,A2(FX2,SX2)	64	RX3
ABL	R1,D2(X2)	65	RX1,RX2
ABL	R1,A2(FX2,SX2)	65	RX3

Operation

The register specified by R1 contains the fullword element to be added to the list, which is located in memory at the address of the second operand. The number of slots used tally is compared with the number of slots in the list. If the number of slots used equals the number of slots in the list, an overflow condition exists. The element is not added to the list and the overflow flag in the condition code is set.

If the number of slots used tally is less than the number of slots in the list, it is incremented by one, the appropriate pointer is changed, and the element is added to the list. (Refer to Figure 3-4.)

Condition Code

C	V	G	L	
0	0	0	0	Element added successfully
0	1	0	0	List overflow

Programming Notes

These instructions manipulate circular lists as described in the introduction to this chapter.

The second operand location must be on a fullword boundary.

The ATL instruction manipulates the current top pointer in the list. If no overflow occurs, the current top pointer, which points to the last element added to the top of the list, is decremented by one. The element is inserted in the slot pointed to by the new current top pointer. If the current top pointer was zero on entering this instruction, the current top pointer is set to the maximum slot number in the list. This condition is referred to as list wrap.

The ABL instruction manipulates the next bottom pointer. If no overflow occurs, the element is inserted in the slot pointed to by the next bottom pointer, and the next bottom pointer is incremented by one. If the incremented next bottom pointer is greater than the maximum slot number in the list, the next bottom pointer is set to zero. This condition is referred to as list wrap.

For the nonoverflow situation, pointer halfwords in the list header are not manipulated until after the element has been successfully added. This facilitates error recovery in the event of a memory fault.

See examples in the next section.

3.5.41 Remove From List (RTL, RBL)

Remove from Top of List (RTL)
Remove from Bottom of List (RBL)

Assembler Notation		Op-Code	Format
RTL	R1,D2(X2)	66	RX1,RX2
RTL	R1,A2(FX2,SX2)	66	RX3
RBL	R1,D2(X2)	67	RX1,RX2
RBL	R1,A2(FX2,SX2)	67	RX3

Operation

The element removed from the list replaces the contents of the register specified by R1. The list is located at the address of the second operand. If, at the start of the instruction execution, the number of slots used tally is zero, then the list is already empty and the instruction terminates with the overflow flag set in the condition code. This condition is referred to as list underflow; in this case, R1 is undefined. If underflow does not occur, the appropriate pointer is changed, the element is extracted and placed in the register specified by R1, and the number of slots used tally is decremented by one.

Condition Code

C	V	G	L	
0	0	0	0	List now empty
0	0	1	0	List is not yet empty
0	1	0	0	List was already empty

Programming Notes

These instructions manipulate circular lists as described in the introduction to this chapter.

The second operand location must be on a fullword boundary.

In the case of list underflow, the contents of the register specified by R1 are unchanged.

The RTL instruction manipulates the current top pointer. If no underflow occurs, the current top pointer points to the element to be extracted. The element is extracted and placed in the register specified by R1. The current top pointer is incremented by one and compared to the maximum slot number. If the current top pointer is greater than the maximum slot number, the current top pointer is set to zero. This condition is referred to as list wrap.

The RBL instruction manipulates the next bottom pointer. If no underflow occurs, and the next bottom pointer is zero, it is set to the maximum slot number (list wrap); otherwise, it is decremented by one, and the element now pointed to is extracted and placed in the register specified by R1.

For the nonunderflow situation, pointer halfwords in the list header are not manipulated until after the element has been successfully removed. The register specified by R1 is not modified until the header has been updated. This facilitates error recovery in the event of a memory fault.

Examples: List Instructions (ATL, ABL, RTL, RBL)

The following are examples of the use of the four list processing instructions.

The original list is normally set up as shown in Figure 3-7.

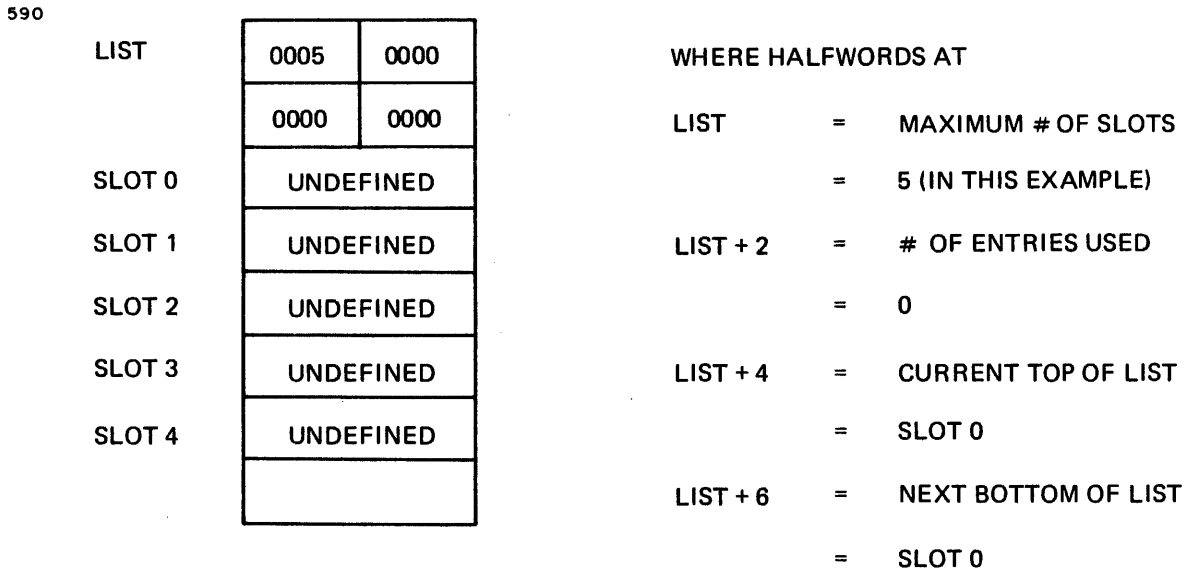


Figure 3-7 List Processing Instructions

Assembler Notation		Results and Comments
LIS	REG0,0	
STH	REG0,LIST+2	INITIALIZE NUMBER OF ENTRIES USED TO 0
ST	REG0,LIST+4	INITIALIZE POINTERS TO 0
LIS	REG1,1	REGISTERS 1 THROUGH 6 CONTAIN
LIS	REG2,2	1 THROUGH 6 RESPECTIVELY
LIS	REG3,3	
LIS	REG4,4	
LIS	REG5,5	
LIS	REG6,6	
STH	REG5,LIST	TOTAL NUMBER OF ENTRIES = 5

```
REF1    ATL REG1,LIST
```

LIST	0005	0001
	0004	0000
SLOT 0	UNDEFINED	
SLOT 1	UNDEFINED	
SLOT 2	UNDEFINED	
SLOT 3	UNDEFINED	
SLOT 4	0000	0001

(List Wrap)

Condition Code = 0000
Current Top Pointer = Slot 4
Next Bottom Pointer = Slot 0

```
REF2    ATL REG2,LIST
```

LIST	0005	0002
	0003	0000
SLOT 0	UNDEFINED	
SLOT 1	UNDEFINED	
SLOT 2	UNDEFINED	
SLOT 3	0000	0002
SLOT 4	0000	0001

Condition Code = 0000
Current Top Pointer = Slot 3
Next Bottom Pointer = Slot 0

```
REF3   ATL REG3,LIST
```

LIST	0005	0003
	0002	0000
SLOT 0	UNDEFINED	
SLOT 1	UNDEFINED	
SLOT 2	0000	0003
SLOT 3	0000	0002
SLOT 4	0000	0001

Condition Code = 0000
Current Top Pointer = Slot 2
Next Bottom Pointer = Slot 0

```
REF4   ABL REG4,LIST
```

LIST	0005	0004
	0002	0001
SLOT 0	0000	0004
SLOT 1	UNDEFINED	
SLOT 2	0000	0003
SLOT 3	0000	0002
SLOT 4	0000	0001

Condition Code = 0000
Current Top Pointer = Slot 2
Next Bottom Pointer = Slot 1

```
REF5    ABL REG5,LIST
```

LIST	0005	0005
	0002	0002
SLOT 0	0000	0004
SLOT 1	0000	0005
SLOT 2	0000	0003
SLOT 3	0000	0002
SLOT 4	0000	0001

Condition Code = 0000
Current Top Pointer = Slot 2
Next Bottom Pointer = Slot 2

```
REF6    ABL REG6,LIST
```

LIST	0005	0005
	0002	0002
SLOT 0	0000	0004
SLOT 1	0000	0005
SLOT 2	0000	0003
SLOT 3	0000	0002
SLOT 4	0000	0001

Condition Code = 0100 (List overflow)
Current Top Pointer = Slot 2
Next Bottom Pointer = Slot 2

```
REF7   RTL REG7,LIST
```

LIST	0005	0004
	0003	0002
SLOT 0	0000	0004
SLOT 1	0000	0005
SLOT 2 X	0000	0003
SLOT 3	0000	0002
SLOT 4	0000	0001

(REG7) = 0000 0003
Condition Code = 0010
Current Top Pointer = Slot 3
Next Bottom Pointer = Slot 2

```
REF8   RBL REG8,LIST
```

LIST	0005	0003
	0003	0001
SLOT 0	0000	0004
SLOT 1 X	0000	0005
SLOT 2 X	0000	0003
SLOT 3	0000	0002
SLOT 4	0000	0001

(REG8) = 0000 0005
Condition Code = 0010
Current Top Pointer = Slot 3
Next Bottom Pointer = Slot 1

NOTE

X equals entry removed from list, and is not accessible through further manipulation by list instructions.

```
REF9    RTL REG9,LIST
```

LIST	0005	0002
	0004	0001
SLOT 0	0000	0004
SLOT 1 X	0000	0005
SLOT 2 X	0000	0003
SLOT 3 X	0000	0002
SLOT 4	0000	0001

(REG9) = 0000 0002
Condition Code = 0010
Current Top Pointer = Slot 4
Next Bottom Pointer = Slot 1

```
REF10   RBL REG10,LIST
```

LIST	0005	0001
	0004	0000
SLOT 0 X	0000	0004
SLOT 1 X	0000	0005
SLOT 2 X	0000	0003
SLOT 3 X	0000	0002
SLOT 4	0000	0001

(REG10) = 0000 0004
Condition Code = 0010
Current Top Pointer = 4
Next Bottom Pointer = 0

NOTE

X equals entry removed from list, and is not accessible through further manipulation by list instructions.

REF11 RTL REG11,LIST

LIST	0005	0000
	0000	0000
SLOT 0 X	0000	0004
SLOT 1 X	0000	0005
SLOT 2 X	0000	0003
SLOT 3 X	0000	0002
SLOT 4 X	0000	0001

(REG11) = 0000 0001
Condition Code = 0000 (List is now empty)
Current Top Pointer = 0
Next Bottom Pointer = 0

REF12 RTL REG12,LIST

LIST	0005	0000
	0000	0000
SLOT 0 X	0000	0004
SLOT 1 X	0000	0005
SLOT 2 X	0000	0003
SLOT 3 X	0000	0002
SLOT 4 X	0000	0001

(REG12) = UNDEFINED
Condition Code = 0100 (List was
Current Top Pointer = 0 already empty)
Next Bottom Pointer = 0

NOTE

X equals entry removed from list, and is not accessible through further manipulation by list instructions.

CHAPTER 4
BRANCHING

4.1 INTRODUCTION

In normal operations, the processor executes instructions in sequential order. The branch instructions allow this sequential mode of operation to be varied, so that programs can loop, transfer control to subroutines, or make decisions based on the results of previous operations.

4.2 OPERATIONS

The second operand of a branch instruction is the address of the memory location to which control is transferred. The address may be contained in a register or it may be specified in the instruction as the second operand address or as a displacement.

4.2.1 Decision Making

The conditional branch instructions permit the program to make decisions based on some result. In these instructions, the R1 field contains a 4-bit mask, M1, which is tested by ANDing it with the condition code. The result of the test determines whether the branch is taken, or the next sequential instruction is executed.

The following examples show previous condition code, mask specified in a branch instruction, and the result of the test on which the branch or no branch decision is made.

Condition Code	Mask(M1)	Result of Test	(True/ False)	Branch True Taken	Branch False Taken
0000	0010	0000	(False)	No	Yes
0001	1010	0000	(False)	No	Yes
1001	1000	1000	(True)	Yes	No
0100	0100	0100	(True)	Yes	No
1010	0010	0010	(True)	Yes	No
0010	0011	0010	(True)	Yes	No
0010	0000	0000	(False)	No	Yes

4.2.2 Subroutine Linkage

The branch and link instructions allow branching to subroutines in such a way that a return address is passed to the subroutine. For these instructions, the address of the memory location immediately following the branch instruction is saved in the register specified by R1.

4.3 BRANCH INSTRUCTION FORMATS

The branch instructions use the Register-to-Register (RR), the Short Form (SF), and the Register and Indexed Storage (RX) formats.

4.4 BRANCH INSTRUCTIONS

The instructions described in this section are:

BFC	Branch on False Condition
BFCR	Branch on False Condition Register
BFBS	Branch on False Condition Backward Short
BFFS	Branch on False Condition Forward Short
BTC	Branch on True Condition
BTCR	Branch on True Condition Register
BTBS	Branch on True Condition Backward Short
BTFS	Branch on True Condition Forward Short
BAL	Branch and Link
BALR	Branch and Link Register
BXLE	Branch on Index Low or Equal
BXH	Branch on Index High

4.4.1 Branch on True (BTC, BTCR, BTBS, BTFS)

Branch on True Condition (BTC)
Branch on True Condition Register (BTCR)
Branch on True Condition Backward Short (BTBS)
Branch on True Condition Forward Short (BTFS)

Assembler Notation		Op-Code	Format
BTC	M1,D2(X2)	42	RX1,RX2
BTC	M1,A2(FX2,SX2)	42	RX3
BTCR	M1,R2	02	RR
BTBS	M1,N	20	SF
BTFS	M1,N	21	SF

Operation

The condition code of the Program Status Word (PSW) is tested for the conditions specified by the mask field, M1. If any conditions tested are found to be true, a branch is taken to the second operand location. If none of the conditions tested is found to be true, the next sequential instruction is executed.

Condition Code

Unchanged

Programming Notes

In the RR format, the branch address is contained in the register specified by R2.

In the SF format, the N field contains the number of halfwords to be added to or subtracted from the current location counter to obtain the branch address.

In the RR and RX formats, the branch address must be located on a halfword boundary.

Example: BTC

Assembler Notation		Machine Code	Comments
LH	R1,X'100'	4810 0100	Load halfword (X'1234') located at X'100'. Condition code is set to CVGL = 0010. Mask is 3, i.e., M1=0011. Perform logical AND between CVGL and M1, i.e., 0010 AND 0011. The result is 0010, i.e., true; therefore, a branch is taken to LOC.
BTC	3,LOC	4230 ABC0	

4.4.2 Branch on False (BFC, BFCR, BFBS, BFFS)

Branch on False Condition (BFC)
Branch on False Condition Register (BFCR)
Branch on False Condition Backward Short (BFBS)
Branch on False Condition Forward Short (BFFS)

Assembler Notation		Op-Code	Format
BFC	M1,D2(X2)	43	RX1,RX2
BFC	M1,A2(FX2,SX2)	43	RX3
BFCR	M1,R2	03	RR
BFBS	M1,N	22	SF
BFFS	M1,N	23	SF

Operation

The condition code of the PSW is tested for the conditions specified in the mask field, M1. If all conditions tested are found to be false, a branch is taken to the second operand location. If any of the conditions tested is found to be true, the next sequential instruction is executed.

Condition Code

Unchanged

Programming Notes

In the RR format, the branch address is contained in the register specified by R2.

In the SF format, the N field contains the number of halfwords to be added to or subtracted from the current location counter to obtain the branch address.

In the RR and RX formats, the branch address must be located on a halfword boundary.

Example: BFC

Assembler Notation		Machine Code	Comments
LCS	R1,2	2512	(R1) = FFFFFFFE. Condition code, CVGL = 0001 mask is 1001. Perform logical AND between mask and CVGL, i.e., 1001 AND 0001. The result is 0001, i.e., true; therefore, a branch is not taken to LOC.
BFC	9,LOC	4390 AEC0	

4.4.3 Branch and Link (BAL, BALR)

Branch and Link (BAL)
Branch and Link Register (BALR)

Assembler Notation		Op-Code	Format
BAL	R1,D2(X2)	41	RX1,RX2
BAL	R1,A2(FX2,SX2)	41	RX3
BALR	R1,R2	01	RR

Operation

The address of the next sequential instruction is saved in the register specified by R1, and a branch is taken to the second operand address.

Condition Code

Unchanged

Programming Notes

The second operand location must be on a halfword boundary.

The branch address is calculated before the register specified by R1 is changed. R1 may specify the same register as X2, FX2, SX2, or R2.

Example: BAL

The following example illustrates the use of the BAL instruction. This instruction causes contrcl to be transferred to a subroutine called SUBROUT. After completion of the subroutine, the linking register is used to branch back to the next sequential instruction after the BAL; i.e., the instruction labeled RETURN.

	Label	Assembler Notation	Comments
	BEGIN	BAL REG4,SUBROUT	TRANSFER TO SUBROUT
MAIN	RETURN	XR R6,R6	
PROG		STH R6,LAB+4	
		:	
		:	
	SUBROUT	LHL R8,LCC	THE RETURN ADDRESS OF THE SUBROUTINE IS IN REG4
SUBROUTINE		AHI R8,10	
		:	
		:	
	RTNEND	BR REG4	RETURN TO XR INST.

NOTE

The linking register (REG4 in the example) should not be used within the subroutine.

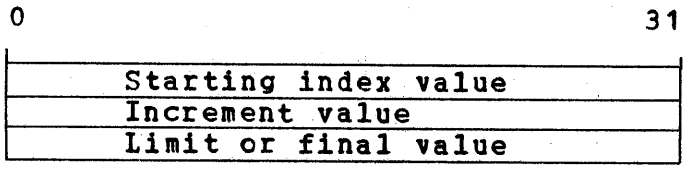
Result of BAL Instruction

(REG4) = Address of instruction at SUBROUT
Condition Code unchanged

4.4.4 Branch on Index Low or Equal (BXLE)

Assembler Notation		Op-Code	Format
BXLE	R1,D2(X2)	C1	RX1,RX2
BXLE	R1,A2(FX2,SX2)	C1	RX3

Set Up



Before execution of this instruction, the register specified by R1 must contain a starting index value. The register specified by R1+1 must contain an increment value. The register specified by R1+2 must contain a comparand (limit or final value). All values may be signed.

Operation

Execution of this instruction causes the increment value to be added to the index value, creating a new index value. The result is compared logically to the limit or final value. If the new index value is less than or equal to the limit value, a branch is taken to the second operand location. If the new index value is greater than the limit value, the next sequential instruction is executed.

Condition Code

Unchanged

Programming Notes

The incremented index value replaces the contents of the register specified by R1.

Any three consecutive registers of the same set may be used by this instruction as specified by R1. These registers may be 6, 7, 8; or 14, 15, 0; or 15, 0, 1, etc.

The second operand location must be on a halfword boundary.

The branch address is calculated before incrementing the starting index value contained in the register specified by R1.

R1 may specify the same register as X2, FX2 or SX2.

Example: BXLE

Transfer 10 bytes in memory starting at the memory location labeled BUF0 to the memory location labeled BUF1.

```
Label    Assembler Notation          Comments

         LIS     REG3,0          (REG3)=STARTING INDEX VALUE=0
         LIS     REG4,1          (REG4)=INCREMENT VALUE
         LIS     R5,9            (REG5)=FINAL VALUE=9

AGAIN    LB      REG0,BUF0(R3)   (REG0)=1 BYTE FROM BUF0
         STB     REG0,BUF1(R1)   COPY 1 BYTE TO BUF1
LABEL    BXLE    R3,AGAIN        IF (REG3)>(REG5),DONE
         .
         .
         .
BUF0     DS      10
BUF1     DS      10
```

Result of BXLE Instruction

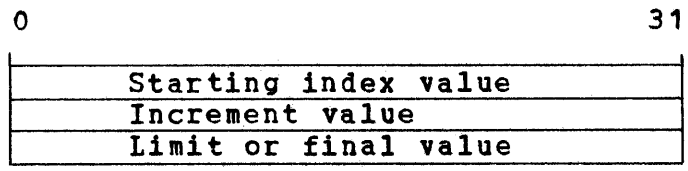
Code between the instructions labeled AGAIN and LABEL is executed ten times.

Condition Code unchanged by BXLE instruction
(REG3) = 0000000A
(REG4) = 00000001
(REG5) = 00000009

4.4.5 Branch on Index High (BXH)

Assembler Notation		Op-Code	Format
BXH	R1,D2(X2)	C0	RX1,RX2
BXH	R1,A2(FX2,SX2)	C0	RX3

Set Up



Before execution of this instruction, the register specified by R1 must contain a starting index value. The register specified by R1+1 must contain an increment value. The register specified by R1+2 must contain a comparand (limit or final value). All values may be signed.

Operation

Execution of this instruction causes the increment value to be added to the index value, creating a new index value. The result is logically compared to the limit or final value. If the new index value is greater than the limit value, a branch is taken to the second operand location. If the new index value is less than or equal to the limit value, the next sequential instruction is executed.

Condition Code

Unchanged

Programming Notes

The incremented index value replaces the contents of the register specified by R1.

Any three consecutive registers of the same set may be used by this instruction as specified by R1. These registers may be 6, 7, 8; 14, 15, 0; or 15, 0, 1, etc.

The second operand location must be on a halfword boundary.

The branch address is calculated before incrementing the starting index value contained in the register specified by R1.

R1 may specify the same register as X2, FX2 or SX2.

Example: BXH

The following example shows how to set up a counter (1-9) using the BXH instruction:

```
Label     Assembler Notation          Comment

          LIS     REG1,1            (REG1)=0000 0001 (INDEX)
          LIS     REG2,1            (REG2)=0000 0001 (INCREMENT)
          LIS     REG3,9            (REG3)=0000 0009 (COMPARAND)
BEGIN     BXH     REG1,LABEL        COMPARE INDEX WITH COMPARAND
          LH      R6,CCUNT
          .
          .
          .
          B       BEGIN             BRANCH TO BXH INSTRUCTION
LABEL     LA      R8,RTN            EXIT FROM BXH
          ST      R8,MEM
```

Result of BXH Instruction

Code between the instructions labeled BEGIN and LABEL is executed nine times.

Condition Code Unchanged by BXH instruction
(REG1) = 0000 000A
(REG2) = 0000 0001
(REG3) = 0000 0009

4.5 EXTENDED BRANCH MNEMONICS

The CAL assembler supports 47 extended branch mnemonics that generate the branch op-code (true or false conditional) and the condition code mask required. The programmer must supply the second operand address (symbolic or absolute). In the case of Short Format (SF) branch instructions, the second operand branch address must be within 15 halfwords of the current location counter. The CAL assembler determines the backward or forward relationship of the second operand address and generates the appropriate operation code.

The instructions described in this section are:

BC	Branch on Carry
BCR	Branch on Carry Register
BCS	Branch on Carry Short
BNC	Branch on No Carry
BNCR	Branch on No Carry Register
BNCS	Branch on No Carry Short
BE	Branch on Equal
BER	Branch on Equal Register
BES	Branch on Equal Short
BNE	Branch on Not Equal
BNER	Branch on Not Equal Register
BNES	Branch on Not Equal Short
BL	Branch on Low
BLR	Branch on Low Register
BLS	Branch on Low Short
BNL	Branch on Not Low
BNLR	Branch on Not Low Register
BNLS	Branch on Not Low Short
BM	Branch on Minus
BMR	Branch on Minus Register
BMS	Branch on Minus Short
BNM	Branch on Not Minus
BNMR	Branch on Not Minus Register
BNMS	Branch on Not Minus Short
BP	Branch on Plus
BPR	Branch on Plus Register
BPS	Branch on Plus Short
BNP	Branch on Not Plus
BNPR	Branch on Not Plus Register
BNPS	Branch on Not Plus Short

BO	Branch on Overflow
BOR	Branch on Overflow Register
BOS	Branch on Overflow Short
BNO	Branch on No Overflow
BNOR	Branch on No Overflow Register
BNOS	Branch on No Overflow Short
BZ	Branch on Zero
BZR	Branch on Zero Register
BZS	Branch on Zero Short
BNZ	Branch on Not Zero
BNZR	Branch on Not Zero Register
BNZS	Branch on Not Zero Short
B	Branch (Unconditional)
BR	Branch Register (Unconditional)
BS	Branch Short (Unconditional)
NOP	No Operation
NOPR	No Operation Register

4.5.1 Branch on Carry (BC, BCR, BCS)

Branch on Carry (BC)
Branch on Carry Register (BCR)
Branch on Carry Short (BCS)

Assembler Notation		Op-Code+M1	Format
BC	D2(X2)	428	RX1,RX2
BC	A2(FX2,SX2)	428	RX3
BCR	R2	028	RR
BCS	A	208(Backward) 218(Forward)	SF

Operation

If the Carry (C) flag in the condition code is set, a branch is taken to the second operand location. If the C flag is zero, the next sequential instruction is executed.

Condition Code

Unchanged

Programming Notes

The branch address must be located on a halfword boundary.

In the RR format, the branch address is contained in the register specified by R2.

Example: BCS

Assembler Notation			Machine Code	Comments
SHIFT	SLLS	R9,1	1191	Register 9 is shifted
	BCS	SHIFT	2081	left until the first zero bit is shifted out of position 0.

4.5.2 Branch on No Carry (BNC, BNCR, BNCS)

Branch on No Carry (BNC)
Branch on No Carry Register (BNCR)
Branch on No Carry Short (BNCS)

Assembler Notation		Op-Code+M1	Format
BNC	D2(X2)	438	RX1,RX2
BNC	A2(FX2,SX2)	438	RX3
BNCR	R2	038	RR
BNCS	A	228 (Backward)	SF
		238 (Forward)	

Operation

If the Carry (C) flag in the condition code is zero, a branch is taken to the second operand location. If the C flag is set, the next sequential instruction is executed.

Condition Code

Unchanged

Programming Notes

The branch address must be located on a halfword boundary.

In the RR format, the branch address is contained in the register specified by R2.

4.5.3 Branch on Equal (BE, BER, BES)

Branch on Equal (BE)
Branch on Equal Register (BER)
Branch on Equal Short (BES)

Assembler Notation		Op-Code+M1	Format
BE	D2(X2)	433	RX1,RX2
BE	A2(FX2,SX2)	433	RX3
BER	R2	033	RR
BES	A	223 (Backward)	SF
		233 (Forward)	

Operation

If the G flag and the L flag are both zero in the condition code, a branch is taken to the second operand location. If either flag is set, the next sequential instruction is executed.

Condition Code

Unchanged

Programming Notes

The branch address must be located on a halfword boundary.

In the RR format, the branch address is contained in the register specified by R2.

Example: BE

Assembler Notation		Machine Code	Comments
CLHI	R4,X'23'	C540 0023	If R4 contains X'23',
BE	CPTIN	4330 0A00	a branch is taken to location X'A00'. Otherwise, the next sequential instruction is executed.

4.5.4 Branch on Not Equal (BNE, BNER, BNES)

Branch on Not Equal (BNE)
Branch on Not Equal Register (BNER)
Branch on Not Equal Short (BNES)

Assembler Notation		Op-Code+M1	Format
BNE	D2(X2)	423	RX1,RX2
BNE	A2(FX2,SX2)	423	RX3
BNER	R2	023	RR
BNES	A	203 (Backward)	SF
		213 (Forward)	

Operation

If the G flag or the L flag is set in the condition code, a branch is taken to the second operand location. If both flags are zero, the next sequential instruction is executed.

Condition Code

Unchanged

Programming Notes

The branch address must be located on a halfword boundary.

In the RR format, the branch address is contained in the register specified by R2.

4.5.5 Branch on Low (BL, BLR, BLS)

Branch on Low (BL)
Branch on Low Register (BLR)
Branch on Low Short (BLS)

Assembler Notation		Op-Code+M1	Format
BL	D2(X2)	428	RX1,RX2
BL	A2(FX2,SX2)	428	RX3
BLR	R2	028	RR
BLS	A	208 (Backward) 218 (Forward)	SF

Operation

If the Carry (C) flag in the condition code is set, a branch is taken to the second operand address. If the C flag is zero, the next sequential instruction is executed.

Condition Code

Unchanged

Programming Notes

The branch address must be located on a halfword boundary.

In the RR format, the branch address is contained in the register specified by R2.

Example: BL

Assembler Notation		Machine Code	Comments
CLHI	R1,X'FF'	C510 00FF	(R1) is compared to X'00FF'. If (R1) is less than X'00FF', a branch is taken to memory location X'0A00'.
BL	RESTART	4280 0A00	

4.5.6 Branch on Not Low (BNL, BNLR, BNLS)

Branch on Not Low (BNL)
Branch on Not Low Register (BNLR)
Branch on Not Low Short (BNLS)

Assembler Notation		Op-Code+M1	Format
BNL	D2(X2)	438	RX1, RX2
BNL	A2(FX2,SX2)	438	RX3
BNLR	R2	038	RR
BNLS	A	228 (Backward)	SF
		238 (Forward)	

Operation

If the Carry (C) flag in the condition code is zero, a branch is taken to the second operand address. If the C flag is set, the next sequential instruction is executed.

Condition Code

Unchanged

Programming Notes

The branch address must be located on a halfword boundary.

In the RR format, the branch address is contained in the register specified by R2.

4.5.7 Branch on Minus (BM, BMR, BMS)

Branch on Minus (BM)
Branch on Minus Register (BMR)
Branch on Minus Short (BMS)

Assembler Notation		Op-Code+M1	Format
BM	D2(X2)	421	RX1,RX2
BM	A2(FX2,SX2)	421	RX3
BMR	R2	021	RR
BMS	A	201 (Backward)	SF
		211 (Forward)	

Operation

If the Less Than (L) flag in the condition code is set, a branch is taken to the second operand location. If the L flag is zero, the next sequential instruction is executed.

Condition Code

Unchanged

Programming Notes

The branch address must be located on a halfword boundary.

In the RR format, the branch address is contained in the register specified by R2.

Example: BM

Assembler Notation		Machine Code	Comments
SIS	R3,1	2631	If (R3) is less than 0 after the subtraction, a branch is taken to X'10A0'.
BM	CONTINUE	4210 10A0	

4.5.8 Branch on Not Minus (BNM, BNMR, BNMS)

Branch on Not Minus (BNM)
Branch on Not Minus Register (BNMR)
Branch on Not Minus Short (BNMS)

Assembler Notation		Op-Code+M1	Format
BNM	D2(X2)	431	RX1,RX2
BNM	A2(FX2,SX2)	431	RX3
BNMR	R2	031	RR
BNMS	A	221 (Backward)	SF
		231 (Forward)	

Operation

If the Less Than (L) flag in the condition code is zero, a branch is taken to the second operand location. If the L flag is set, the next sequential instruction is executed.

Condition Code

Unchanged

Programming Notes

The branch address must be located on a halfword boundary.

In the RR format, the branch address is contained in the register specified by R2.

4.5.9 Branch on Plus (BP, BPR, BPS)

Branch on Plus (BP)
Branch on Plus Register (BPR)
Branch on Plus Short (BPS)

Assembler Notation		Op-Code+M1	Format
BP	D2(X2)	422	RX1,RX2
BP	A2(FX2,SX2)	422	RX3
BPR	R2	022	RR
BPS	A	202 (Backward)	SF
		212 (Forward)	

Operation

If the Greater Than (G) flag in the condition code is set, a branch is taken to the second operand location. If the G flag is zero, the next sequential instruction is executed.

Condition Code

Unchanged

Programming Notes

The branch address must be located on a halfword boundary.

In the RR format, the branch address is contained in the register specified by R2.

4.5.10 Branch on Not Plus (BNP, BNPR, BNPS)

Branch on Not Plus (BNP)
Branch on Not Plus Register (BNPR)
Branch on Not Plus Short (BNPS)

Assembler Notation		Op-Code+M1	Format
BNP	D2(X2)	432	RX1,RX2
BNP	A2(FX2,SX2)	432	RX3
BNPR	R2	032	RR
BNPS	A	222 (Backward)	SF
		232 (Forward)	

Operation

If the Greater Than (G) flag in the condition code is zero, a branch is taken to the second operand location. If the G flag is set, the next sequential instruction is executed.

Condition Code

Unchanged

Programming Notes

The branch address must be located on a halfword boundary.

In the RR format, the branch address is contained in the register specified by R2.

4.5.11 Branch on Overflow (BO, BOR, BOS)

Branch on Overflow (BO)
Branch on Overflow Register (BCR)
Branch on Overflow Short (BOS)

Assembler Notation		Op-Code+M1	Format
BO	D2(X2)	424	RX1,RX2
BO	A2(FX2,SX2)	424	RX3
BOR	R2	024	RR
BOS	A	204 (Backward)	SF
		214 (Forward)	

Operation

If the Overflow (V) flag in the condition code is set, a branch is taken to the second operand location. If the V flag is zero, the next sequential instruction is executed.

Condition Code

Unchanged

Programming Notes

The branch address must be located on a halfword boundary.

In the RR format, the branch address is contained in the register specified by R2.

4.5.12 Branch on No Overflow (BNO, BNOR, BNOS)

Branch on No Overflow (BNO)
Branch on No Overflow Register (BNOR)
Branch on No Overflow Short (BNOS)

Assembler Notation		Op-Code+M1	Format
BNO	D2(X2)	434	RX1,RX2
BNO	A2(FX2,SX2)	434	RX3
BNOR	R2	034	RR
BNOS	A	224 (Backward)	SF
		234 (Forward)	

Operation

If the Overflow (V) flag in the condition code is zero, a branch is taken to the second operand location. If the V flag is set, the next sequential instructicn is executed.

Condition Code

Unchanged

Programming Notes

The branch address must be located on a halfword boundary.

In the RR format, the branch address is contained in the register specified by R2.

4.5.13 Branch on Zero (BZ, BZR, BZS)

Branch on Zero (BZ)
Branch on Zero Register (BZR)
Branch on Zero Short (BZS)

Assembler Notation		Op-Code+M1	Format
BZ	D2(X2)	433	RX1,RX2
BZ	A2(FX2,SX2)	433	RX3
BZR	R2	033	RR
BZS	A	223 (Backward)	SF
		233 (Forward)	

Operation

If the G and L flags are both zero in the condition code, a branch is taken to the second operand location. If the G or L flag is set, the next sequential instruction is executed.

Condition Code

Unchanged

Programming Notes

The branch address must be located on a halfword boundary.

In the RR format, the branch address is contained in the register specified by R2.

4.5.14 Branch on Not Zero (BNZ, BNZR, BNZS)

Branch on Not Zero (BNZ)
Branch on Not Zero Register (BNZR)
Branch on Not Zero Short (BNZS)

Assembler Notation		Op-Code+M1	Format
BNZ	D2(X2)	423	RX1,RX2
BNZ	A2(FX2,SX2)	423	RX3
BNZR	R2	023	RR
BNZS	A	203 (Backward)	SF
		213 (Forward)	

Operation

If the G or L flag in the condition code is set, a branch is taken to the second operand address. If the G and L flags are both zero, the next sequential instruction is executed.

Condition Code

Unchanged

Programming Notes

The branch address must be located on a halfword boundary.

In the RR format, the branch address is contained in the register specified by R2.

4.5.15 Branch (Unconditional) (B, BR, BS)

Branch (Unconditional) (B)
Branch Register (Unconditional) (BR)
Branch Short (Unconditional) (BS)

Assembler Notation		Op-Code+M1	Format
B	D2(X2)	430	RX1,RX2
B	A2(FX2,SX2)	430	RX3
BR	R2	030	RR
BS	A	220 (Backward)	SF
		230 (Forward)	

Operation

A branch is unconditionally taken to the second operand address.

Condition Code

Unchanged

Programming Notes

The branch address must be located on a halfword boundary.

In the RR format, the branch address is contained in the register specified by R2.

This instruction is assembled as a Branch on False Condition instruction, with no condition specified (M1=0). Therefore, the branch test is always false and the branch is always taken.

Example: B

Assembler Notation	Machine Code	Comments
B CPTIN	4300 0A00	An unconditional branch is taken to location X'0A00'.

4.5.16 No Operation (NOP, NOPR)

No Operation (NOP)
No Operation Register (NOPR)

Assembler Notation		Op-Code+M1	Format
NOP	D2(X2)	420	RX1,RX2
NOP	A2(FX2,SX2)	420	RX3
NOPR	R2	020	RR

Operation

The next sequential instruction is executed.

Condition Code

Unchanged

Programming Notes

D2(X2) or A2(FX2,SX2) and R2 are ignored and usually equal zero (0).

This instruction is assembled as a Branch on True Condition instruction with no condition specified (M1=0); therefore, no branch is taken and the next instruction is fetched and executed.

Example: NOP,NOPR

Assembler Notation		Machine Code	Comments
NOP	0(0,0)	4200 4000 0000	No operation
NOP	0	4200 0000	No operation
NOPR		0200	No operation

CHAPTER 5
FIXED-POINT ARITHMETIC

5.1 INTRODUCTION

Fixed-point arithmetic instructions provide a complete set of operations for calculating addresses and indices, for counting, and for general purpose fixed-point arithmetic.

5.2 FIXED-POINT DATA FORMATS

There are three formats for fixed-point data: the halfword, the fullword, and the doubleword. In each of these formats, the most significant bit (bit 0) is the sign bit. The remaining 15, 31 or 63 bits represent the magnitude. (See Figure 5-1.)

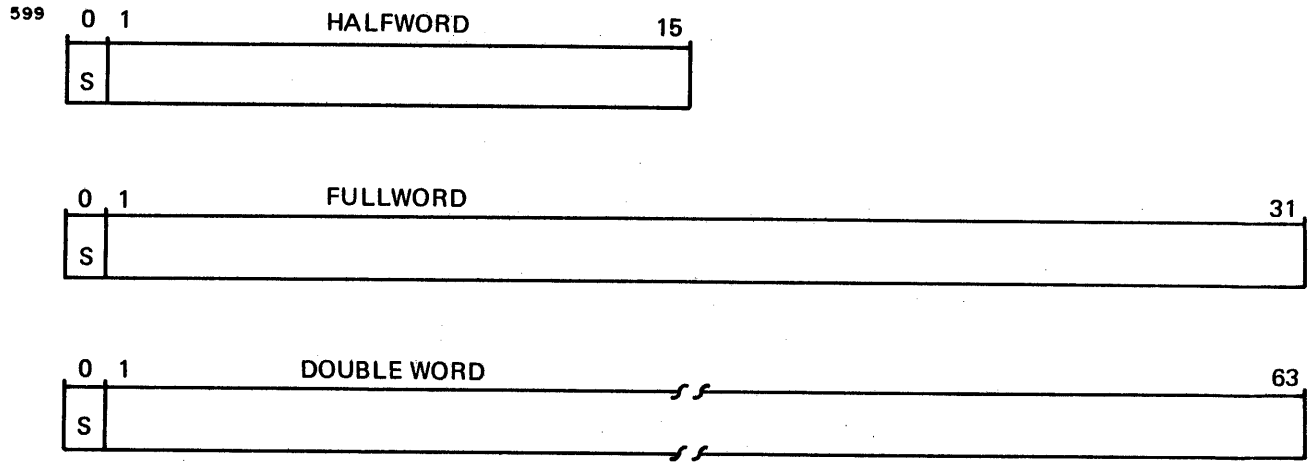


Figure 5-1 Fixed-Point Data Formats

Positive values are represented in true binary form with a sign bit of zero. Negative values are represented in two's complement form with a sign bit of one. To change the sign of a number, the two's complement of the number may be produced by subtracting the number from zero. Another way would be to:

1. Change all zeros to ones, and all ones to zeros.
2. Add one.

5.3 FIXED-POINT NUMBER RANGE

Fixed-point numbers represent integers. Table 5-1 shows relations between different formats, along with decimal values.

TABLE 5-1 FIXED-POINT FORMAT RELATIONS

600

DOUBLE WORD	FULLWORD	HALFWORD	DECIMAL
8000000000000000 (MOST NEGATIVE)			-9 223 372 036 854 775 808
	80000000 (MOST NEGATIVE)		-2 147 483 648
		8000 (MOST NEGATIVE)	-32 768
FFFFFFFFFFFFFFFF	FFFFFFFF	FFFF (LEAST NEGATIVE)	-1
0000000000000000	00000000	0000	0
0000000000000001	00000001	0001 (LEAST POSITIVE)	1
		7FFF (MOST POSITIVE)	32 767
	7FFFFFFF (MOST POSITIVE)		2 147 483 647
7FFFFFFFFFFFFFFF (MOST POSITIVE)			9 223 372 036 854 775 807

5.4 OPERATIONS

Fixed-point instructions include both fullword and halfword operations. Fullword operations take place (a) between the contents of two general registers; (b) between the contents of a general register and a fullword stored in memory; or (c) between the contents of a general register and a fullword obtained from the instruction stream. Fullword multiply produces a doubleword result which is contained in two adjacent registers. Fullword divide operates on doubleword data contained in two adjacent registers.

Halfword operations take place between a fullword contained in one of the general registers and a halfword contained in memory. Before the operation is started, the halfword in memory is expanded to a fullword by propagating the most significant bit (sign bit) into the high order bits of the fullword. The halfword multiply and divide instructions are exceptions to this rule.

5.5 CONDITION CODE

All fixed-point arithmetic instructions, except multiply and divide, affect the condition code to indicate the outcome of the operation on the 32-bit result.

In fixed-point add and subtract operations, the arguments are represented in two's complement form; therefore, all bits, including sign, participate in forming the result. Consequently, the occurrence of a carry or borrow has no real arithmetic significance.

For example, an add operation between a minus one (FFFF FFFF) and a plus two (0000 0002) produces the correct result of plus one (0000 0001) and a carry. The condition code is set to 1010 (C = 1 and G = 1). Carry means that the complete result, which in this case would have been 1 0000 0001, would not fit in 32 bits.

An overflow occurs when the result does not fit in 31 bits. Note that bit zero must be reserved for the sign of the result. For example, adding one to the largest positive fixed-point value produces an overflow:

```
 7FFF FFFF
+0000 0001
=8000 0000
```

The resulting condition code is 0101 (V=1 and L=1).

The result, 8000 0000, is logically correct, but because the sign bit is negative when the result should be positive, the overflow condition exists.

The columns of the condition code table given for each instruction description show the state of the C, V, G and L flags for the possible results.

An 'X' in a condition code column means that the particular flag is not defined, and may be either 0 or 1. Hence, no inference should be drawn by testing that particular flag.

5.6 FIXED-POINT INSTRUCTION FORMATS

The fixed-point instructions use the Register to Register (RR), the Short Form (SF), the Register and Indexed Storage (RX), and the Register and Immediate (RI) instruction formats.

5.7 FIXED-PCINT INSTRUCTICNS

The fixed-point instructions described in this section are:

A	Add
AR	Add Register
AI	Add Immediate
AIS	Add Immediate Short
AH	Add Halfword
AHI	Add Halfword Immediate
AM	Add to Memory
AHM	Add Halfword to Memory
S	Subtract
SR	Subtract Register
SI	Subtract Immediate
SIS	Subtract Immediate Short
SH	Subtract Halfword
SHI	Subtract Halfword Immediate
C	Compare
CR	Compare Register
CI	Compare Immediate
CH	Compare Halfword
CHI	Compare Halfword Immediate
M	Multiply
MR	Multiply Register
MH	Multiply Halfword
MHR	Multiply Halfword Register
D	Divide
DR	Divide Register
DH	Divide Halfword
DHR	Divide Halfword Register
SLA	Shift Left Arithmetic
SLHA	Shift Left Halfword Arithmetic
SRA	Shift Right Arithmetic
SRHA	Shift Right Halfword Arithmetic
CHVR	Convert to Halfword Value Register

5.7.1 Add (A, AR, AI, AIS)

Add (A)
Add Register (AR)
Add Immediate (AI)
Add Immediate Short (AIS)

Assembler Notation		Op-Code	Format
A	R1,D2(X2)	5A	RX1,RX2
A	R1,A2(FX2,SX2)	5A	RX3
AR	R1,R2	0A	RR
AI	R1,I2(X2)	FA	RI2
AIS	R1,N	26	SF

Operation

The second operand is added algebraically to the contents of the register specified by R1. The result of this 32-bit addition replaces the contents of the register specified by R1.

Condition Code

C	V	G	L	
X	0	0	0	Result is zero
X	0	0	1	Result is less than zero
X	0	1	0	Result is greater than zero
X	1	X	X	Arithmetic overflow
1	X	X	X	Carry

Programming Notes

The second operand for the AIS instruction is obtained by expanding the 4-bit data field, N, to a 32-bit fullword by forcing the high order bits to zero.

In the RI2 format, the contents of the index register specified by X2 are added to the 32-bit I2 field to form the fullword second operand.

In the RX formats the second operand must be located on a fullword boundary.

Example: A

Add contents of memory location labeled LAB to the contents of REG4.

1. REG4 contains X'7F341234'
   Fullword in memory at LAB contains X'7F124321'

Assembler Notation	Comments
A REG4,LAB	ADD (LAB) TO (REG4)

Result of A Instruction

(REG4) = X'FE465555'
(LAB) unchanged by this instruction
Condition Code = 0101 (V=1, L=1)

2. REG5 contains X'8000 0001'
   Fullword in memory at LAB contains X'80000002'

Assembler Notation	Comments
A REG5,LAB	ADD (LAB) TO (REG5)

Result of A Instruction

(REG5) = X'00000003'
(LAB) unchanged by this instruction
Condition Code = 1110 (C=1, V=1, G=1)

5.7.2 Add Halfword (AH, AHI)

Add Halfword (AH)
Add Halfword Immediate (AHI)

Assembler Notation		Op-Code	Format
AH	R1,D2(X2)	4A	RX1,RX2
AH	R1,A2(FX2,SX2)	4A	RX3
AHI	R1,I2(X2)	CA	RI1

Operation

The 16-bit second operand is expanded to a 32-bit fullword by propagating the most significant bit through bits 0:15 of the fullword. The fullword operand is added to the fullword contents of the register specified by R1. The result replaces the contents of the register specified by R1.

Condition Code

C	V	G	L	
X	0	0	0	Result is zero
X	0	0	1	Result is less than zero
X	0	1	0	Result is greater than zero
X	1	X	X	Arithmetic overflow
1	X	X	X	Carry

Programming Notes

In the RX formats, the second operand must be located on a halfword boundary.

In the RI1 format, the 16-bit I2 field is extended to a fullword by propagating the sign bit through bits 0:15. The contents of the index register specified by X2 are then added to form the fullword second operand.

Example: AH

This example adds the halfword at memory location labeled LAB to the contents of register 4.

1. REG4 contains X'00230002'
   Halfword at memory location LAB contains X'FFFF'

Assembler Notation	Comments
AH REG4,LAB	ADD (LAB) TO (REG4)

Result of AH Instruction

(REG4) = X'00230001'
(LAB) unchanged by this instruction
Condition Code = 1010 (C=1, G=1)

2. REG5 contains X'FFFF FFF5'
   LAB contains X'FFF2'

Assembler Notation	Comments
AH REG5,LAB	ADD (LAB) TO (REG5)

Result of AH Instruction

(REG5) = 'FFFF FFE7'
(LAB) unchanged by this instruction
Condition Code = 1001 (C=1, L=1)

5.7.3 Add to Memory (AM)

Assembler Notation		Op-Code	Format
AM	R1,D2(X2)	51	RX1,RX2
AM	R1,A2(FX2,SX2)	51	RX3

Operation

The first operand contained in the register specified by R1 is added algebraically to the fullword second operand. The result replaces the fullword second operand in memory. The contents of the register specified by R1 are not changed.

Condition Code

C	V	G	L	
X	0	0	0	Result is zero
X	0	0	1	Result is less than zero
X	0	1	0	Result is greater than zero
X	1	X	X	Arithmetic overflow
1	X	X	X	Carry

Programming Note

The second operand must be located on a fullword boundary.

Example: AM

1. Add contents of register 8 to memory location labeled LOC:

   REG8 contains X'00000008'
   Fullword in memory at LOC contains X'034289AB'

Assembler Notation		Comments
AM	REG8,LOC	ADD (REG8) TO (LOC)

Result of AM Instruction

(REG8) unchanged by this instruction
(LCC) = X'034289B3'
Condition Code = 0010 (G=1)

2. Add contents of register 7 to memory location labeled LOC:

REG7 contains X'7F341234'
Fullword in memory at LOC contains X'7F124321'

Assembler Notation	Comments
AM REG7,LCC	ADD (REG7) TO (LOC)

Result of AM Instruction

(REG7) unchanged by this instruction
(LCC) = X'FE465555'
Condition Code = 0101 (V=1, L=1)

5.7.4 Add Halfword to Memory (AHM)

Assembler Notation		Op-Code	Format
AHM	R1,D2(X2)	61	RX1,RX2
AHM	R1,A2(FX2,SX2)	61	RX3

Operation

The halfword second operand is added algebraically to the least significant 16 bits (bits 16:31) of the register specified by R1. The 16-bit result replaces the contents of the memory location specified by the effective address of the second operand. The contents of the register specified by R1 are not changed.

Condition Code

C	V	G	L	
X	0	0	0	Result is zero
X	0	0	1	Result is less than zero
X	0	1	0	Result is greater than zero
X	1	X	X	Arithmetic overflow
1	X	X	X	Carry

Programming Notes

The second operand must be located on a halfword boundary.

The condition code settings are based on the halfword result.

Example: AHM

This example adds the contents of register 5 to the contents of memory location LAB.

1. REG5 contains X'00230002'
   Halfword in memory at LAB contains X'FFFF'

Assembler Notation		Comments
AHM	REG5,LAB	ADD (REG5) TO (LAB)

Result of AHM Instruction

(REG5) unchanged by this instruction
(LAB) = 0001
Condition Code = 1010 (C=1, G=1)

2. REG6 contains X'FFFF FFF5'
LAB contains X'FFF2'

Assembler Notation	Comments
AHM REG6,LAB	ADD (REG6) TO (LAB)

Result of AHM Instruction

(REG6) unchanged by this instruction
(LAB) = FFE7
Condition Code = 1001 (C=1, L=1)

5.7.5 Subtract (S, SR, SI, SIS)

Subtract (S)
Subtract Register (SR)
Subtract Immediate (SI)
Subtract Immediate Short (SIS)

Assembler Notation		Op-Code	Format
S	R1,D2(X2)	5B	RX1,RX2
S	R1,A2(FX2,SX2)	5B	RX3
SR	R1,R2	0B	RR
SI	R1,I2(X2)	FB	RI2
SIS	R1,N	27	SF

Operation

The fullword second operand is subtracted algebraically from the contents of the register specified by R1. The result replaces the contents of the register specified by R1.

Condition Code

C	V	G	L	
X	0	0	0	Result is zero
X	0	0	1	Result is less than zero
X	0	1	0	Result is greater than zero
X	1	X	X	Arithmetic overflow
1	X	X	X	Borrow

Programming Notes

The second operand for the SIS instruction is obtained by expanding the 4-bit data field, N, to a 32-bit fullword by forcing the high order bits to zero.

In the RI2 format, the contents of the index register specified by X2 are added to the 32-bit I2 field to form the fullword second operand.

In the RX formats, the second operand must be located on a fullword boundary.

Examples:

This example subtracts the fullword at memory location LOC from the contents of register 9.

1. REG9 contains X'44444444'
   LCC contains X'44444444'

Assembler Notation		Comments
S	REG9,LOC	SUBTRACT (LOC) FROM (REG9)

Result of S Instruction

(REG9) = 0
(LCC) unchanged by this instruction
Condition Code = 0000

2. REG9 contains X'23456789'
   LCC contains X'FFFF4321'

Assembler Notation		Comments
S	REG9,LOC	SUBTRACT (LOC) FROM (REG9)

Result of S Instruction

(REG9) = 23462368
(LOC) unchanged by this instruction
Condition Code = 1010 (C=1, G=1)

5.7.6 Subtract Halfword (SH, SHI)

Subtract Halfword (SH)
Subtract Halfword Immediate (SHI)

Assembler Notation		Op-Code	Format
SH	R1,D2(X2)	4B	RX1,RX2
SH	R1,A2(FX2,SX2)	4E	RX3
SHI	R1,I2(X2)	CB	RI1

Operation

The 16-bit second operand is expanded to a 32-bit fullword by propagating the most significant bit through bits 0:15. This fullword is subtracted from the contents of the register specified by R1. The result replaces the contents of the register specified by R1.

Condition Code

C	V	G	L	
X	0	0	0	Result is zero
X	0	0	1	Result is less than zero
X	0	1	0	Result is greater than zero
X	1	X	X	Arithmetic overflow
1	X	X	X	Borrow

Programming Notes

In the RX formats, the second operand must be located on a halfword boundary.

In the RI1 format, the 16-bit I2 field is extended to a fullword by propagating the sign bit through bits 0:15. The contents of the index register specified by X2 are then added to form the fullword second operand.

Example: SH

This example subtracts the halfword at memory location LOC from the contents of register 9.

1. REG9 contains X'00123456'
   LOC contains X'FFF4'

Assembler Notation	Comments
SH REG9,LOC	SUBTRACT (LOC) FROM (REG9)

Result of SH Instruction

(REG9) = 00123462
(LOC) unchanged by this instruction
Condition Code = 1010

2. REG9 contains X'FFFF4567'
   LOC contains X'2345'

Assembler Notation	Comments
SH REG9,LOC	SUBTRACT (LOC) FROM (REG9)

Result of SH Instruction

(REG9) = FFFF2222
(LOC) unchanged by this instruction
Condition Code = 0001

5.7.7 Compare (C, CR, CI)

Compare (C)
Compare Register (CR)
Compare Immediate (CI)

Assembler Notation		Op-Code	Format
C	R1,D2(X2)	59	RX1,RX2
C	R1,A2(FX2,SX2)	59	RX3
CR	R1,R2	09	RR
CI	R1,I2(X2)	F9	RI2

Operation

The first operand contained in the register specified by R1 is compared algebraically to the 32-bit second operand. The result is indicated by the condition code setting. Neither operand is changed.

Condition Code

C	V	G	L	
0	X	0	0	First operand is equal to second operand
1	X	0	1	First operand is less than second operand
0	X	1	0	First operand is greater than second operand

Programming Notes

In the RX formats, the second operand must be located on a fullword boundary.

The state of the V flag is undefined.

Example: C

This example compares the contents of register 3 to the contents of the fullword in memory location LAB.

REG3 contains X'44567894'
Fullword at LAB contains X'04321243'

Assembler Notation		Comments
C	REG3,LAB	COMPARE (REG3) TO (LAB)

Result of C Instruction

(REG3) unchanged by this instruction
(LAB) unchanged by this instruction
Condition Code = 0010 (G=1)

5.7.8 Compare Halfword (CH, CHI)

Compare Halfword (CH)
Compare Halfword Immediate (CHI)

Assembler Notation		Op-Code	Format
CH	R1,D2(X2)	49	RX1,RX2
CH	R1,A2(FX2,SX2)	49	RX3
CHI	R1,I2(X2)	C9	RI1

Operation

The halfword second operand is expanded to a fullword by propagating the most significant bit through bits 0:15. The first operand, the contents of the register specified by R1, is compared algebraically to the effective second operand. The result is indicated by the condition code setting. Neither operand is changed.

Condition Code

C	V	G	L	
0	X	0	0	First operand is equal to second operand
1	X	0	1	First operand is less than second operand
0	X	1	0	First operand is greater than second operand

Programming Notes

In the RX formats, the second operand must be located on a halfword boundary.

In the RI1 format, the 16-bit I2 field is extended to a fullword by propagating the sign bit through bits 0:15. The contents of the index register specified by X2 are then added to form the fullword second operand.

Condition code settings are based on the fullword comparison. The state of the V flag is undefined.

Example: CH

This example compares the contents of register 8 to the halfword at LAB.

REG8 contains X'F4567891'
Halfword at LAB contains X'3123'

Assembler Notation	Comments
CH REG8,LAB	COMPARE (REG8) TO (LAB)

Result of CH Instruction

(REG8) unchanged by this instruction
(LAB) unchanged by this instruction
Condition Code = 1001 (C=1, V=1)

5.7.9 Multiply (M, MR)

Multiply (M)
Multiply Register (MR)

Assembler Notation		Op-Code	Format
M	R1,D2(X2)	5C	RX1,RX2
M	R1,A2(FX2,SX2)	5C	RX3
MR	R1,R2	1C	RR

Operation

The fullword first operand contained in the register specified by R1+1 is multiplied by the fullword second operand. The 64-bit result is stored in the registers specified by R1 and R1 + 1. The sign of the result is determined by the rules of algebra.

Condition Code

Unchanged

Programming Notes

The R1 field of these instructions must specify an even numbered register. If the R1 field of these instructions is odd, the result is undefined.

In the RX formats, the second operand must be located on a fullword boundary.

The most significant bits of the result are placed in the register specified by R1; the least significant bits are placed in the register by R1+1.

Example: M

This example multiplies the contents of register 9 by the contents of memory location LOC and places the result in registers 8 and 9 (64 bits).

REG8 contains unknown data
REG9 contains X'00002431'
Fullword at location LOC contains X'43120000'

Assembler Notation	Comments
M REG8,LOC	MULTIPLY (REG9) BY (LOC)

Result of M Instruction

REG8 and REG9 together contain the result
(REG8, REG9) = 0000 097B, 5E72 0000
(LOC) unchanged by this instruction
Condition Code unchanged by this instruction

Example: MR

This example multiplies the contents of register 9 by the contents of register 8 and places the result in registers 8 and 9 (64 bits).

REG8 contains X'00010000'
REG9 contains X'12345678'

Assembler Notation	Comments
MR REG8,REG8	MULTIPLY (REG9) BY (REG8)

Result of MR Instruction

REG8 and REG9 together contain the result
(REG8, REG9) = 0000 1234, 5678 0000
Condition Code unchanged by this instruction

5.7.10 Multiply Halfword (MH, MHR)

Multiply Halfword (MH)
Multiply Halfword Register (MHR)

Assembler Notation		Op-Code	Format
MH	R1,D2(X2)	4C	RX1,RX2
MH	R1,A2(FX2,SX2)	4C	RX3
MHR	R1,R2	0C	RR

Operation

The first operand, contained in bits 16:31 of the register specified by R1, is multiplied by the 16-bit second operand, taken from memory or from bits 16:31 of the register specified by R2. Both operands are 16-bit signed two's complement values. The 32-bit result replaces the contents of the register specified by R1. The sign of the result is determined by the rules of algebra.

Condition Code

Unchanged

Programming Note

In the RX formats, the second operand must be located on a halfword boundary.

Example: MH

This example multiplies the halfword contents of register 8 by the halfword in memory location LAB.

REG8 contains X'ABCD 0045'
Halfword at memory location LAB contains X'8674'

Assembler Notation	Comments
MH REG8,LAB	MULTIPLY LEAST SIGNIFICANT HALF OF (REG8) BY (LAB)

Result of MH Instruction

(REG8) = FFDF3D44
(LAB) unchanged by this instruction
Condition Code unchanged by this instruction

Example: MHR

This example multiplies the halfword contents of register 11 by the halfword ccntents of register 4.

REG11 contains X'37210004'
REG4 contains X'FFFF0307'

Assembler Notation	Comments
MHR REG11,REG4	MULTIPLY LS HALF OF (REG11) BY LS HALF OF (REG4)

Result of MHR Instruction

(REG11) = 00000C1C
(REG4) unchanged by this instruction
Condition Code unchanged by this instruction

5.7.11 Divide (D, DR)

Divide (D)
Divide Register (DR)

Assembler Notation		Op-Code	Format
D	R1,D2(X2)	5D	RX1,RX2
D	R1,A2(FX2,SX2)	5D	RX3
DR	R1,R2	1D	RR

Operation

The 64-bit signed dividend contained in the two registers specified by R1 and R1+1 is divided by the signed fullword second operand. The 32-bit signed remainder replaces the contents of the register specified by R1. The signed 32-bit quotient replaces the contents of the register specified by R1+1.

The sign of the quotient is determined by the rules of algebra. The sign of the remainder is the same as the sign of the dividend.

Condition Code

Unchanged

Programming Notes

The R1 field of these instructions must specify an even numbered register. If the R1 field of these instructions is odd, the result is undefined.

The most significant bits of the dividend must be contained in the register specified by R1. The least significant bits of the dividend must be contained in the register specified by R1+1.

In the RX formats, the second operand must be located on a fullword boundary.

If the divisor is equal to zero, the instruction is not executed, the operand registers remain unchanged, and the arithmetic fault interrupt is taken.

If the value of the quotient is more positive than X'7FFFFFFF' or more negative than X'80000000', quotient overflow is said to occur. If quotient overflow occurs, the operand registers remain unchanged, and the arithmetic fault interrupt is taken.

Example: D

This example divides the contents of registers 8 and 9 by the fullword contents of memory location LOC.

1. REG8 contains X'12345678' = Most significant half of dividend
   REG9 contains X'98765432' = Least significant half of dividend
   LOC contains X'34343434' = Divisor

Assembler Notation	Comments
D REG8,LOC	DIVIDE (REG8,9) BY (LOC)

Result of D Instruction

(REG8) = 1E1E1E1E = Remainder
(REG9) = 59455459 = Quotient
(LOC) unchanged by this instruction
Condition Code unchanged by this instruction

2. REG8 contains X'FFFF1234' = Most significant half of dividend
   REG9 contains X'00000000' = Least significant half of dividend
   LCC contains X'12345678' = Divisor

Assembler Notation	Comments
D REG8,LOC	DIVIDE (REG8,9) BY (LOC)

Result of D Instruction

(REG8) = F250D9E0 = Remainder
(REG9) = FFF2EFFC = Quotient
LCC unchanged by this instruction
Condition Code unchanged by this instruction

3. REG8 contains X'43657898' = Most significant half of dividend
   REG9 contains X'12123456' = Least significant half of dividend
   LOC contains X'00000000' = Divisor

Assembler Notation	Comments
D REG8,LOC	DIVIDE (REG8,9) BY (LOC)

Result of D Instruction

Division by zero causes arithmetic fault to be taken. Operands and condition code remain unchanged by this instruction.

4. REG8 contains X'80000000' = Most significant half of dividend
   REG9 contains X'00000001' = Least significant half of dividend
   LOC contains X'00000001' = Divisor

Assembler Notation	Comments
D REG8,LOC	DIVIDE (REG8,9) BY (LOC)

Result of D Instruction

Quotient overflow causes arithmetic fault to be taken. Operands and condition code remain unchanged by this instruction.

Example: DR

This example divides the contents of registers 8 and 9 by the contents of register 2.

REG8 contains X'FFFFFFFF' = Most significant half of dividend
REG9 contains X'FFFFFFFD' = Least significant half of dividend
REG2 contains X'FFFFFFFE' = Divisor

Assembler Notation	Comments
DR REG8,REG2	DIVIDE (REG8,9) BY (REG2)

Result of DR instruction

(REG8) = FFFFFFFF = Remainder
(REG9) = 00000001 = Quotient
(REG2) unchanged by this instruction
Condition Code unchanged by this instruction

5.7.12 Divide Halfword (DH, DHR)

Divide Halfword (DH)
Divide Halfword Register (DHR)

Assembler Notation		Op-Code	Format
DH	R1,D2(X2)	4D	RX1,RX2
DH	R1,A2(FX2,SX2)	4D	RX3
DHR	R1,R2	0D	RR

Operation

The 32-bit signed dividend contained in the register specified by R1 is divided by the 16-bit signed second operand. The 16-bit signed remainder is copied to R1 (bits 16:31) and the halfword value is converted to a fullword value. The 16-bit signed quotient is copied to the register specified by R1 + 1 after conversion to a fullword value.

The sign of the quotient is determined by the rules of algebra. The sign of the remainder is the same as the sign of the dividend.

Condition Code

Unchanged

Programming Notes

In the RX formats, the second operand must be located on a halfword boundary. In the RR format, the second operand is taken from bits 16:31 of the register specified by R2.

If the divisor is equal to zero, the instruction is not executed, the operand registers remain unchanged, and the arithmetic fault interrupt is taken.

If the value of the quotient is more positive than X'7FFF' or more negative than X'8000', quotient overflow is said to occur. If quotient overflow occurs, the operand registers remain unchanged, and the arithmetic fault interrupt is taken.

Example: DH

This example divides the contents of register 7 by the halfword contents of memory location LCC.

1. REG7 contains X'0000 0054' = Dividend
   LOC contains X'0008' = Divisor

Assembler Notation	Comments
DH REG7,LOC	DIVIDE (REG7) BY (LOC)

Result of DH Instruction

(REG7) = 0000 0004 = Remainder
(REG8) = 0000 000A = Quctient
(LOC) unchanged by this instruction
Condition Code unchanged by this instruction

2. REG7 contains X'1234 5678' = Dividend
   LOC contains X'0000' = Divisor

Assembler Notation	Comments
DH REG7,LOC	DIVIDE (REG7) BY (LOC)

Result of DH Instruction

Division by zero causes arithmetic fault to be taken. Operands and condition code remain unchanged by this instruction.

3. REG7 contains X'8000 0002' = Dividend
   LOC contains X'0001'

Assembler Notation	Comments
DH REG7,LCC	DIVIDE (REG7) BY (LOC)

Result of DH Instruction

Quotient overflow causes arithmetic fault to be taken. Operands and condition code remain unchanged by this instruction.

5.7.13 Shift Left Arithmetic (SLA)

Assembler Notation	Op-Code	Format
SLA R1,I2(X2)	EE	RI1

Operation

Bits 1:31 of the first operand, contained in the register specified by R1, are shifted left the number of places specified by the second operand. The sign bit (bit 0) remains unchanged. Bits shifted out of position 1 are shifted through the carry flag and then lost. The last bit shifted remains in the carry flag. Zeros are shifted into position 31.

Condition Code

C	V	G	L	
X	0	0	0	Result is zero
X	0	0	1	Result is less than zero
X	0	1	0	Result is greater than zero

Programming Notes

The state of the C flag indicates the state of the last bit shifted.

The shift count is specified by the least significant five bits of the second operand. The maximum shift count is 31.

A shift of zero places causes the condition code to be set in accordance with the value contained in the register specified by R1. The C flag is zero in this case.

Example: SLA

This example shifts the bits in register 5 left by the number specified by the second operand.

REG5 contains X'80005647'

Assembler Notation	Comments
SLA REG5,4	SHIFT (REG5) LEFT 4 PLACES

Result of SLA Instruction

(REG5) = 80056470
Condition Code = 0001 (L=1)

5.7.14 Shift Left Halfword Arithmetic (SLHA)

Assembler Notation	Op-Code	Format
SLHA R1,I2(X2)	CF	RI1

Operation

Bits 17:31 of the register specified by R1 are shifted left the number of places specified by the second operand. Bit 16 of the register, the halfword sign bit, remains unchanged. Bits shifted out of position 17 are shifted through the C flag and then lost. The last bit shifted remains in the C flag. Zeros are shifted into position 31. Bits 0:15 of the first operand register remain unchanged.

Condition Code

C	V	G	L	
X	0	0	0	Result is zero
X	0	0	1	Result is less than zero
X	0	1	0	Result is greater than zero

Programming Notes

The condition code settings are based on the halfword (bits 16:31) result.

The state of the C flag indicates the state of the last bit shifted.

The shift count is specified by the least significant four bits of the second operand. The maximum shift count is 15.

A shift of zero places causes the condition code to be set in accordance with the halfword value contained in bits 16:31 of the register specified by R1. The C flag is zero in this case.

5.7.15. Shift Right Arithmetic (SRA)

Assembler Notation	Op-Code	Format
SRA R1,I2(X2)	EE	RI1

Operation

Bits 1:31 of the first operand, contained in the register specified by R1, are shifted right the number of places specified by the second operand. The sign bit (bit 0) remains unchanged and is propagated right as many positions as specified by the second operand. Bits shifted out of position 31 are shifted through the C flag and lost. The last bit shifted remains in the C flag.

Condition Code

C	V	G	L	
X	0	0	0	Result is zero
X	0	0	1	Result is less than zero
X	0	1	0	Result is greater than zero

Programming Notes

The state of the C flag indicates the state of the last bit shifted.

The shift count is specified by the least significant five bits of the second operand. The maximum shift count is 31.

A shift of zero places causes the condition code to be set in accordance with the value contained in the register specified by R1. The C flag is zero in this case.

Example: SRA

This example shifts the contents of register 9 right the number of places specified by the second operand.

REG9 contains X'800004256'

Assembler Notation	Comments
SRA REG9,8	SHIFT (REG9) RIGHT 8 PLACES

Result of SRA Instruction

(REG9) = X'FF800042'
Condition Code = 0001 (L=1)

5.7.16 Shift Right Halfword Arithmetic (SRHA)

Assembler Notation	Op-Code	Format
SRHA R1,I2 (X2)	CE	RI1

Operation

Bits 17:31 of the register specified by R1 are shifted right the number of places specified by the second operand. Bit 16 of the register, the halfword sign bit, remains unchanged and is propagated right the number of positions specified by the second operand. Bits shifted out of position 31 are shifted through the C flag and lost. The last bit shifted remains in the C flag. Bits 0:15 of the first operand register remain unchanged.

Condition Code

C	V	G	L	
X	0	0	0	Result is zero
X	0	0	1	Result is less than zero
X	0	1	0	Result is greater than zero

Programming Notes

The condition code settings are based on the halfword (bits 16:31) result.

The state of the C flag indicates the state of the last bit shifted.

The shift count is specified by the least significant four bits of the second operand. The maximum shift count is 15.

A shift of zero places causes the condition code to be set in accordance with the halfword value contained in bits 16:31 of the register specified by R1. The C flag is zero in this case.

5.7.17 Convert to Halfword Value Register (CHVR)

Assembler Notation	Op-Code	Format
CHVR R1,R2	12	RR

Cperation

The halfword second operand, bits 16:31 of the register specified by R2, is expanded to a fullword by propagating the most significant bit (bit 16) through bits 0:15. This fullword replaces the contents of the register specified by R1.

Condition Code

C	V	G	L	
X	X	0	0	Result is zero
X	X	0	1	Result is less than zero
X	X	1	0	Result is greater than zero
X	1	X	X	Source operand cannot be represented by a 16-bit signed number
1	X	X	X	Carry flag was set in previous condition code
0	X	X	X	Carry flag was zero in previous condition code

Programming Notes

The V flag is set when bit 15 of the second operand is not the same as bit 16 of the second operand. The G and L flags reflect the algebraic value of bits 16:31 of the second operand.

Execution of this instruction following halfword operations guarantees the same results as those cbtained if the program were run cn a 16-bit machine. For example, if location A in memory contains the halfword value of X'7FFF' (decimal 32767) then,

```
LH      R1,A      R1 contains X'00007FFF'
AIS     R1,1      R1 contains X'00008000'
```

Following the add operation, the condition code is:

C	V	G	L
0	0	1	0

indicating a result greater than zero, which is correct for fullword operations. If the same sequence were executed on a 16-bit processor, as:

```
LH      R1,A      R1 contains X'7FFF'
AIS     R1,1      R1 contains X'8000'
```

Following this, the condition code in the halfword processor is:

C	V	G	L
0	1	0	1

indicating overflow and a negative result. Going back to the original sequence and adding the Convert to Halfword Value Register instruction produces the following:

```
LH      R1,A      R1 contains X'00007FFF'
AIS     R1,1      R1 contains X'00008000'
CHVR    R1,R1     R1 contains X'FFFF8000'
```

Following this sequence, the condition code is:

C	V	G	L
0	1	0	1

which is identical to that of the 16-bit processor, and can be tested in the same manner.

CHAPTER 6
FLOATING-POINT ARITHMETIC (OPTIONAL)

6.1 INTRODUCTION

Floating-point arithmetic instructions provide a means for rapid handling of scientific data expressed as floating-point numbers. Single-precision and double-precision floating-point instructions, as well as mixed mode floating-point instructions, are described in this chapter. The comprehensive set of instructions includes load and store floating-point numbers; add, subtract, multiply, divide and compare two floating-point numbers; convert fixed-point to floating-point and vice versa; and mixed mode operations that translate single-precision to double-precision and vice versa.

Floating-point is a means of representing a quantity in any numbering system. For example, the decimal number 123 (base = 10), can be represented in the following forms:

$$\begin{aligned} 123.0 &\times 10^0 \\ 1.23 &\times 10^2 \\ 0.123 &\times 10^3 \\ 0.0123 &\times 10^4 \end{aligned}$$

In this example, the decimal point moved; this is called a floating-point. In actual floating-point representation, the significant digits are always fractional and are collectively referred to as fractions. The power to which the base number is raised is called the exponent. For example, in the number .45678 x 10^2, 45678 is the fraction and 2 is the exponent. Both the fraction and the exponent can be signed. If we have a floating-point representation such as,

(sign of fraction) (exponent) (fraction)

the following representation applies:

Number	Floating-point			
+32.94	= +.3294 x 10^2	+	+2	3294
-23760000.0	= -.2376 x 10^8	-	+8	2376
+0.000059	= +.59 x 10^{-4}	+	-4	59
-0.0000000092073	= -.92073 x 10^{-8}	-	-8	92073

Large or small numbers can be easily expressed in floating-point, making it ideally suitable for scientific computation. Note the compactness of floating-point notation in the above examples.

Floating-point representation in the processor is similar to the above representation. The differences are:

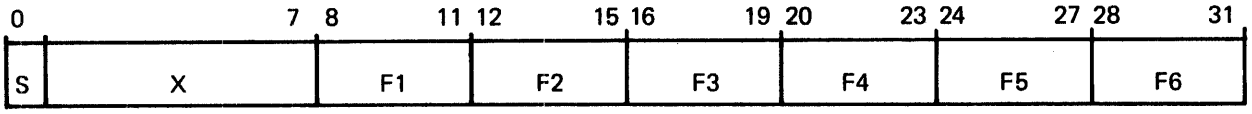
1. Hexadecimal, instead of decimal, numbering system is used.

2. Physical size of the number is limited, therefore the magnitude and precision are limited.

6.2 FLOATING-POINT DATA FORMATS

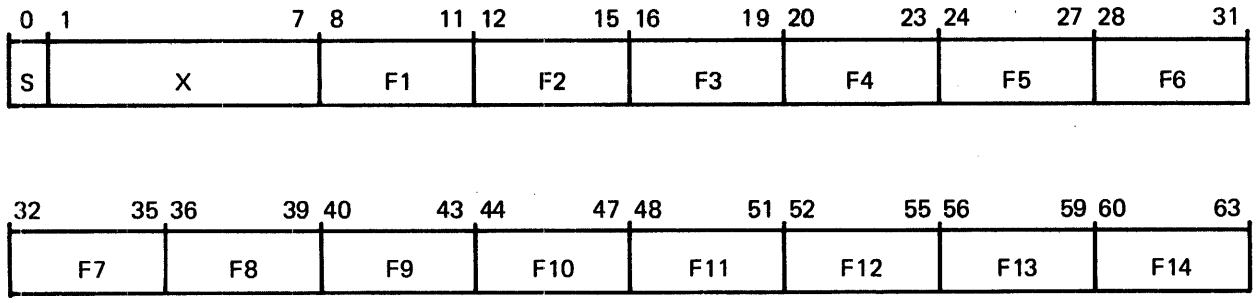
Floating-point numbers occur in one of two formats: single-precision and double-precision. The single-precision format requires a fullword (32 bits). When such a value is contained in memory, it must exist on a fullword address boundary. The sign (S), exponent (X), and fraction (consisting of the digits F1, F2, F3, F4, F5, and F6) fields are designated as follows:

602



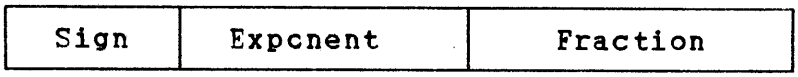
The dcuble-precision format requires a doubleword (64 bits). When two general registers hold a double-precision value, an even/odd pair of general registers must be used. The even-numbered register contains the most significant 32 bits, and the next sequential odd register contains the least significant 32 bits. The sign (S), exponent (X), and fraction (consisting of digits F1 through F14) fields are designated as follows:

603



6.3 FLOATING-POINT NUMBER

In the processor, a floating-point number is represented in the following form:



Sign — The most significant bit of a floating-point number is the sign bit. The sign bit is zero for positive numbers and one for negative numbers. The floating-point value of zero always has a positive sign.

Exponent — The 7-bit field, bits 1:7, is designated as the exponent field. The exponent is expressed in excess-64 notation. The number in this field contains the true value of the exponent plus X'40' (decimal 64). This helps to represent very small magnitudes between 0 and 1. Some of the exponent values are as follows:

Exponent in Excess-64 notation	True exponent in hexadecimal	True exponent in decimal	Multiply fraction by
00	-40	-64	16
3F	-1	-1	16
40	0	0	16
41	1	1	16
7F	3F	63	16

The exponent field for true zero is always 00.

Fraction — The fraction field is 6 hexadecimal digits for single-precision floating-point numbers (thus limiting the precision), and 14 hexadecimal digits for double-precision floating-point numbers. As in any other fraction, the floating-point fraction is expressed with most precision when the most significant hexadecimal digit (not necessarily the most significant bit) is nonzero. The floating-point number with such a fraction is called a normalized floating-point number. In the 3200 processors, normalized numbers are always used to obtain the maximum possible precision. For hexadecimal fraction conversion, refer to Appendix D.

Examples: The following examples illustrate the sign, exponent, and fraction concept of a floating-point number:

Numbers in Hexadecimal integer-fraction notation	Sign-exponent-fraction shown for clarity			Single-precision floating-point numbers
	S	E	F	
+1.3A25678	0	41	13A25678	4113A256
-6.89F2C	1	41	689F2C	C1689F2C
+1A.C39D21	0	42	1AC39021	421AC39D
-3C1DF.82A3	1	45	3C1DF82A3	C53C1DF8
+ABCDEF12.9AC	0	48	ABCDEF129AC	48ABCDEF
+0.0032A9CF2	0	3E	32A9CF2	3E32A9CF
-0.000002C7B5	1	3B	2C7B5	BB2C7B50

6.3.1 Floating-Point Number Range

The range of magnitude (M) of a normalized floating-point number is as follows:

Single precision: $16^{-65} < M \leq (1 - 16^{-6}) * 16^{63}$
Double precision: $16^{-65} < M \leq (1 - 16^{-14}) * 16^{63}$
Approximately for both: $5.4 * 10^{-79} < M < 7.2 * 10^{79}$

Table 6-1 shows the floating-point range in relation to the fixed-point range along with the decimal values.

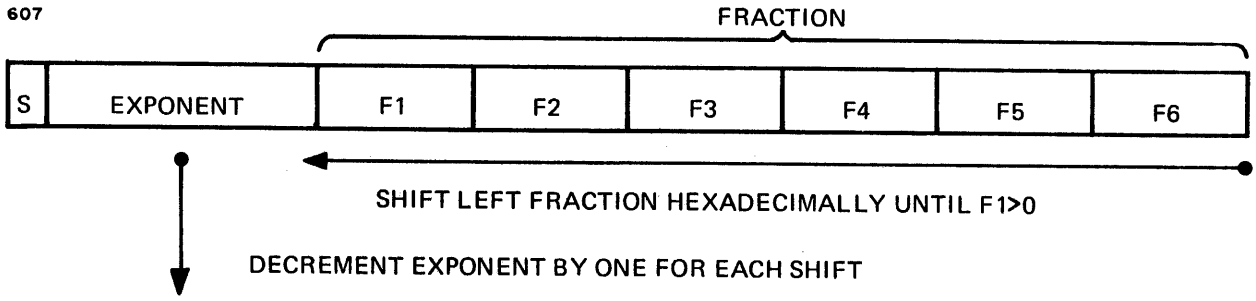
TABLE 6-1 FLOATING/FIXED-POINT RANGES

606-1

FLOATING-POINT NUMBERS		FIXED-POINT INTEGER	DECIMAL NUMBERS
(most negative)	FFFF FFFF		$-7.2* 10^{75}$
	C880 0000	8000 0000 (most negative)	-2 147 483 648
	C110 0000	FFFF FFFF (least negative)	-1
(least negative)	8010 0000		$-5.4*10^{-79}$
(true zero)	0000 0000	0000 0000	0
(least positive)	0010 0000		$+5.4*10^{-79}$
	4110 0000	0000 0001 (least positive)	+1
	487F FFFF	7FFF FFFF (most positive)	+2 147 483 647
(most positive)	7FFF FFFF		$+7.2*10^{75}$

6.3.2 Normalization

Normalization is a process of making nonzero the most significant digit (F1) of the fraction of a floating-point number. In the normalization process, the floating-point fraction is shifted left hexadecimally (i.e., four bits at a time), and its exponent is decremented by one for each hexadecimal shift until the most significant digit (not necessarily the most significant bit) of the fraction is nonzero.



Except for the load instructions, all floating-point operations assume and require normalized operands for consistent results. The load instructions normalize an unnormalized operand.

Example:

	Operands	After normalization
1.	42012345	41123450
2.	21000ABC	1EABC000
3.	C900FE12	C7FE1200
4.	6C000000	00000000 (true zero)
5.	82000A67	00000000 (exponent underflow)

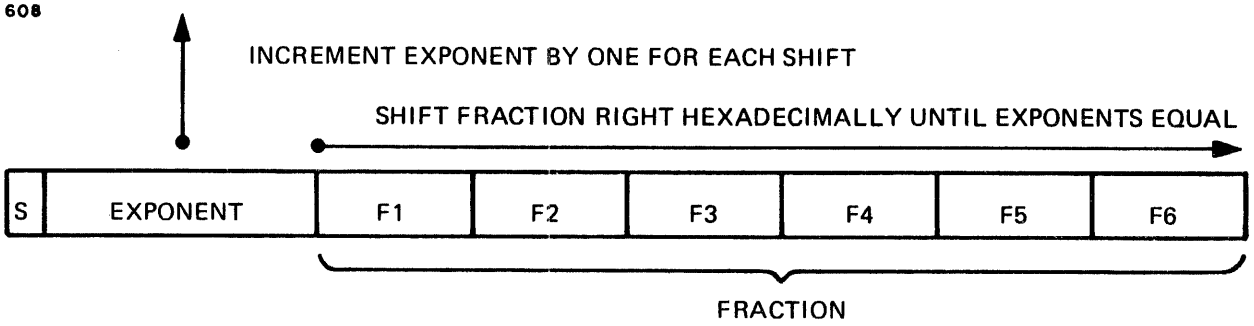
In Example 4, the fraction of the operand is zero. During the normalization process, such a fraction is detected, and the floating-point number is set to true zero.

In Example 5, the exponent of the operand is very small. During the normalization process, the exponent is decremented from 00 to 7F. Such a transition results in exponent underflow, and the floating-point number is set to true zero.

Normalized results are always produced in floating-point operations, assuming the operands are normalized. Results of operations between unnormalized numbers are undefined.

6.3.3 Equalization

Equalization is a process of equalizing exponents of two floating-point numbers. The fraction of the floating-point number with the smaller exponent is shifted right hexadecimally, i.e., four bits at a time, and its exponent is incremented by one for each hexadecimal shift until the two exponents are equal.



During floating-point addition and subtraction, the two floating-point operands are equalized.

Example:

	Floating-point operands	After equalization
1.	43123456 3F789ABC	43123456 43000078
2.	C7FE1234 4956789A	C900FE12 4956789A

In this example, normalized floating-point numbers are shown because addition and subtraction require normalization. If the exponents differ by more than 6 for single-precision or more than 14 for double-precision, the representable significance of the lower exponent floating-point number is lost in the process of equalization. Digits shifted out are shifted through the guard digits and may still have an effect on the result, sum, or difference.

6.3.4 True Zero

A floating-point number is true zero when the exponent and the fraction fields are all zercs; therefore, all data bits must be zero. A zero value always has a positive sign. In general, zero values participate as normal operands in all floating-point operations.

A true zero may be used as an operand. It may also result from an arithmetic operation that caused an exponent underflow, in which case the entire number may be forced to true zero. If an arithmetic operation produces a result whose fraction digits are all zeros (sometimes referred to as loss of significance), the entire number is forced to true zero.

Examples:

Numbers	Operation	Result	Reason
030000AB	Normalize	0000 0000	exponent underflow
41ABCDEF 41ABCDEF	Subtract	0000 0000	loss of significance

6.3.5 Exponent Overflow

In floating-point operations, exponent overflow occurs when a resulting exponent is greater than +63. If overflow occurs, the result register is unchanged. The condition code is set to reflect the overflow situation and the resulting sign. Figure 6-1 illustrates exponent overflow using a line representation of numbers.

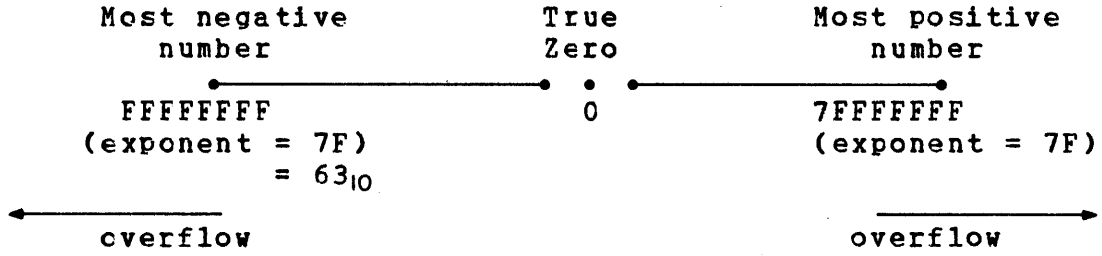


Figure 6-1 Exponent Overflow

If overflow occurs, the V flag in the condition code is set, and an arithmetic fault interrupt is taken. Exponent overflow interrupts cannot be disabled.

6.3.6 Exponent Underflow

The normalization process, during a floating-point operation, may produce an exponent underflow. This underflow occurs when a result exponent is less than -64. Figure 6-2 illustrates exponent underflow using a line representation of numbers.

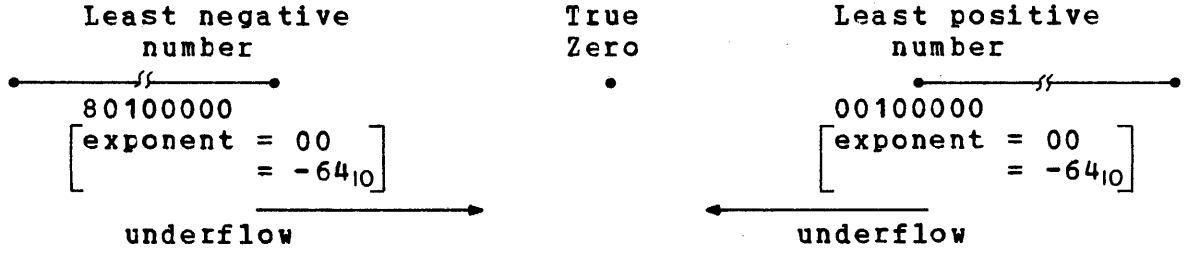


Figure 6-2 Exponent Underflow

If underflow occurs, an arithmetic fault interrupt is taken, if enabled by the current PSW. Both operands remain unchanged. If underflow is disabled by the current PSW, the result is forced to zero (the closest possible answer), the V flag in the condition code is set, and the next sequential instruction is executed.

6.3.7 Guard Digits and R*-Rounding

When an intermediate floating-point result has been formed, it consists of a sign, an exponent, and a fraction field. The fraction field is extended by a number of guard digits containing the least significant fraction digits of the intermediate result. Before the result is copied to a destination, it is rounded to compensate for the loss in the final result of the guard digits.

The rules for the R*-Rounding scheme are:

- If the most significant guard digit is hexadecimal 7 or less, no rounding is performed. (See Example 1.)
- If the most significant guard digit is hexadecimal 8, and all other guard digits are 0, the least significant bit of the final result is forced to 1. (See Example 2.)
- If the most significant guard digit is hexadecimal 8, and another guard digit is nonzero, or if the most significant guard digit is hexadecimal 9 or greater, 1 is added to the fraction field of the final result. (See Example 3.) If this addition produces a carry out of the fraction field (i.e., fraction field was all 1s), the result exponent is incremented by 1, the most significant fraction digit (F1) is set to hexadecimal 1, and all other fraction digits are set to 0. (See Example 4.) Note that exponent overflow could occur as the result of rounding.

Examples of R*-Rounding

	INTERMEDIATE RESULT		FINAL SINGLE-PRECISION RESULT
	Data	Guard Digits	
1.	42ABCD12	32680000	42ABCD12
2.	C1183756	80000000	C1183757
3.	3E265739	80100C0C	3E26573A
4.	41FFFFFF	F0000000	42100000

6.3.8 Conversion from Decimal

To convert a decimal number into the excess-64 notation used internally by the processor, the following steps must be taken:

1. Separate the decimal integer from the decimal fraction:

$$182.375_{10} = (182 + .375)_{10}$$

2. Convert each part to hexadecimal by referring to the integer conversion table and the fraction conversion table in Appendix D.

$$182_{10} = B6_{16} \qquad .375_{10} = .6_{16}$$

3. Combine the hexadecimal integer and fraction:

$$B6.6_{16} = (B6.6X16^{0})_{16}$$

4. Shift the radix point:

$$(B6.6X16^{0})_{16} = (.B66X16^{2})_{16}$$

5. Add 64 (X'40') to the exponent:

$$40_{16} + 2_{16} = 42_{16}$$

6. Convert the exponent field and fractions to binary allowing 1 bit for the sign, 7 bits for exponent field, and 24 or 56 bits for the fraction.

42B66 = 0100 0010 1011 0110 0110 0000 0000 0000

6.4 CONDITION CODE

Most floating-point operations affect the condition code. For each instruction description, the possible condition code settings are shown.

6.5 FLOATING-POINT INSTRUCTIONS

All floating-point instructions are illegal when PSW bit 13 (FLM) is set. Floating-point instructions cannot be executed when the processor is in the Floating-Point Masked (FLM) mode.

Floating-point instructions use the Register to Register (RR), and the Register and Indexed Storage (RX) instruction formats. In all of the RR formats, except for fix and float, the R1 and R2 fields specify one of the floating-point registers. There are eight single-precision floating-point registers and eight double-precision floating-point registers numbered 0, 2, 4, 6, 8, 10, 12, and 14. Floating-point instructions must specify even-numbered floating-point registers, or the results of the instructions are undefined. Except for the FXR, FXDR, LGER, and LGDR instructions, the R1 field always specifies a floating-point register.

Floating-point arithmetic operations, excluding loads and stores, require normalized operands to ensure correct results. If the operands are not normalized, the results of these operations are undefined. Floating-point results are normalized. The floating-point load instructions normalize the floating-point data presented as the second cperand.

The single-precision floating-point instructions described in this section are:

LE	Load Floating-Point
LER	Load Floating-Point Register
LEGR	Load Floating-Point from General Register
LPER	Load Positive Floating-Point Register
LCER	Load Complement Floating-Point Register
LME	Load Floating-Point Multiple
LGER	Load General Register from Floating-Point Register
STE	Store Floating-Point
STME	Store Floating-Point Multiple
AE	Add Floating-Point
AER	Add Floating-Point Register
SE	Subtract Floating-Point
SER	Subtract Floating-Point Register
CE	Compare Floating-Point
CER	Compare Floating-Point Register
ME	Multiply Floating-Point
MER	Multiply Floating-Point Register
DE	Divide Floating-Point
DER	Divide Floating-Point Register
FXR	Fix Register
FLR	Float Register

The double-precision floating-point instructions described in this section are:

LD	Load DPFP
LDR	Load Register DPFP
LDGR	Load DPFP from General Registers
LPDR	Load Positive Register DPFP
LCDR	Load Complement Register DPFP
LMD	Load DPFP Multiple
LGDR	Load General Register from DPFP register
STD	Store DPFP
STMD	Store Multiple DPFP
AD	Add DPFP
ADR	Add Register DPFP
SD	Subtract DPFP
SDR	Subtract Register DPFP
CD	Compare DPFP
CDR	Compare Register DPFP
MD	Multiply DPFP
MDR	Multiply Register DPFP
DD	Divide DPFP
DDR	Divide Register DPFP
FXDR	Fix Register DPFP
FLDR	Float Register DPFP

The mixed mode floating-point instructions described in this section are:

LED	Load SPFP from DPFP
LEDR	Load Register SPFP from DPFP
LDE	Load DPFP from SPFP
LDER	Load Register DPFP from SPFP
STDE	Store DPFP in SPFP

6.5.1 Load Floating-Point (LE, LER, LEGR)

Load Floating-Point (LE)
Load Floating-Point Register (LER)
Load Floating-Point from General Register (LEGR)

Assembler Notation	Op-Code	Format
LE R1,D2(X2)	68	RX1,RX2
LE R1,A2(FX2,SX2)	68	RX3
LER R1,R2	28	RR
LEGR R1,R2	A5	RR

Operation

The floating-point second operand is normalized, if necessary, and placed in the single-precision floating-point register specified by R1.

Condition Code

C	V	G	L	
0	0	0	0	Floating-point result is zero
0	0	0	1	Floating-point result is less than zero
0	0	1	0	Floating-point result is greater than zero
0	1	0	0	Exponent underflow

Programming Notes

If the argument fraction is zero, the entire result is forced to zero, X'0000 0000'.

Normalization can produce exponent underflow. If PSW bit 19 is set, an arithmetic fault interrupt is taken, and the register specified by R1 is unchanged. If an exponent underflow occurs, and bit 19 of the current PSW is zero, no arithmetic fault occurs. Zeros replace the contents of the register specified by R1.

In the RX formats, the second operand must be located on a fullword boundary.

Example: LE

This example normalizes the fullword at memory location LOC and places it in floating-point register 8.

Floating-point REG8 contains unknown data
LOC contains X'4200 1000'

Assembler Notation	Comments
LE REG8,LOC	LOAD FROM LOC AND NORMALIZE

Result of LE Instruction:

(REG8) = X'4010 0000'
(LOC) unchanged by this instruction
Condition Code = 0010

6.5.2 Load Positive Floating-Point Register (LPER)

Assembler Notation	Op-code	Format
LPER R1,R2	13	RR

Operation

The floating-point second operand specified by R2 is forced positive, normalized if necessary, and placed in the single-precision floating-point register specified by R1.

Condition Code

C	V	G	L	
0	0	0	0	Floating-point result is zero
0	0	1	0	Floating-point result is greater than zero
0	1	0	0	Exponent underflow

Programming Notes

If the argument fraction is zero, the entire result is forced to zero, X'0000 0000'.

Normalization can produce exponent underflow. If PSW bit 19 is set, an arithmetic fault interrupt is taken, and the register specified by R1 is unchanged. If an exponent underflow occurs, and bit 19 of the current PSW is zero, no arithmetic fault occurs. Zeros replace the contents of the register specified by R1.

Example:

Floating-point REG6 contains unknown data
Floating-point REG8 contains X'C11921FB'

Assembler Notation	Comments
LPER REG6,REG8	LCAD REG6 WITH POSITIVE OF (REG8)

Result of LPER Instruction:

(REG6) = X'411921FB'
(REG8) unchanged by this instruction
Condition Code = 0010

6.5.3 Load Complement Floating-Point Register (LCER)

Assembler Notation	Op-Code	Format
LCER R1,R2	17	RR

Operation

The sign of the floating-point second operand specified by R2 is complemented. The resulting floating-point number is normalized, if necessary, and placed in the single-precision floating-point register specified by R1.

Condition Code

C	V	G	L	
0	0	0	0	Floating-point result is zero
0	0	0	1	Floating-point result is less than zero
0	0	1	0	Floating-point result is greater than zero
0	1	0	0	Exponent underflow

Programming Notes

If the argument fraction is zero, the entire result is forced to zero, X'0000 0000'.

Normalization can produce expcnent underflow. If PSW bit 19 is set, an arithmetic fault interrupt is taken, and the register specified by R1 is unchanged. If an exponent underflow occurs, and bit 19 of the current PSW is zero, no arithmetic fault occurs. Zeros replace the contents of the register specified by R1.

6.5.4 Load Multiple Floating-Point (LME)

Assembler Notation	Op-Code	Format
LME R1,D2(X2)	72	RX2,RX2
LME R1,A2(FX2,SX2)	72	RX3

Operation

Successive single-precision floating-point registers, starting with the register specified by R1, are loaded from successive fullword memory locations starting with the address of the second operand. The process stops when floating-point register 14 has been loaded.

Condition Code

Unchanged

Programming Notes

Values loaded into the floating-point registers are assumed to be normalized, and no test or adjustment is performed.

The second operand must be located on a fullword boundary.

6.5.5 Load General Register from Floating-Point Register (LGER)

Assembler Notation	Op-Code	Format
LGER R1,R2	15	RR

Operation

The floating-point second operand, contained in the single-precision floating-point register specified by R2, is placed in the general register specified by R1. The second operand is unchanged.

Condition Code

C	V	G	L	
0	0	0	0	Result is zero
0	0	0	1	Result is less than zero
0	0	1	0	Result is greater than zero

6.5.6 Store Floating-Point (STE)

Assembler Notation	Op-Code	Format
STE R1,D2(X2)	60	RX1,RX2
STE R1,A2(FX2,SX2)	60	RX3

Operation

The floating-point first operand, contained in the single-precision floating-point register specified by R1, is placed in the fullword memory location specified by the second operand address. The first operand is unchanged.

Condition Code

Unchanged

Programming Note

The second operand must be located on a fullword boundary.

6.5.7 Store Multiple Floating-Point (STME)

Assembler Notation	Op-Code	Format
STME R1,D2(X2)	71	RX1,RX2
STME R1,A2(FX2,SX2)	71	RX3

Operation

The contents of successive single-precision floating-point registers, starting with the even numbered register specified by R1, are stored in successive fullword memory locations, starting with the address of the second operand. The operation stops when the contents of floating-point register 14 have been stored.

Condition Code

Unchanged

Programming Note

The second operand must be located on a fullword boundary.

6.5.8 Add Floating-Point (AE, AER)

Add Floating-Point (AE)
Add Floating-Point Register (AER)

Assembler Notation		Op-Code	Format
AE	R1,D2(X2)	6A	RX1,RX2
AE	R1,A2(FX2,SX2)	6A	RX3
AER	R1,R2	2A	RR

Operation

The two operand exponents are compared. If the exponents differ, the fraction with the smaller exponent is shifted right hexadecimally (four bits at a time), and its exponent is incremented by one for each hexadecimal shift, until the two exponents are equal. The hexadecimal digits (of four bits each) are shifted through the guard digits for additional precision. If no equalizing shifts are required, the guard digits remain zero. The fractions are then algebraically added. The guard digits participate in this addition.

If the addition of fractions produces a carry, the exponent of the result is incremented by one, and the fraction of the result is shifted right one hexadecimal digit. The carry bit is shifted back into the most significant hexadecimal digit of the fraction, producing a normalized result. This result is then R*-rounded and replaces the contents of the single-precision floating-point register specified by R1.

If the addition of fractions does not produce a carry, the result is normalized, if necessary, and R*-rounded. This result replaces the contents of the single-precision floating-point register specified by R1.

Condition Code

C	V	G	L	
0	0	0	0	Floating-point result is zero
0	0	0	1	Floating-point result is less than zero
0	0	1	0	Floating-point result is greater than zero
0	1	0	1	Exponent overflow, result is less than zero
0	1	1	0	Exponent overflow, result is greater than zero
0	1	0	0	Exponent underflow

Programming Notes

When the addition of the fractions produces a carry, incrementing the exponent of the result by one can produce exponent overflow. In this case, the arithmetic fault interrupt is taken and the contents of the register specified by R1 remain unchanged.

Normalization of the result can produce exponent underflow. If PSW bit 19 is set, an arithmetic fault interrupt is taken, and the register specified by R1 is unchanged. If exponent underflow occurs and bit 19 of the current PSW is zero, no arithmetic fault occurs. Zeros replace the contents of the register specified by R1.

In the RX formats, the second operand must be located on a fullword boundary.

Fastest results occur when the first operand is larger than the second operand.

Example: AE

This example adds the contents of LOC to the contents of floating-point register 8 and places the result in floating-point register 8.

Floating-point REG8 contains X'7EFF FFFF'.
LOC contains X'7EFF FFFF'

Assembler Notation	Comments
AE REG8,LOC	ADD (LOC) TO (REG8)

Result of AE Instruction

(Floating-point REG8) = 7F1F FFFF
(LOC) unchanged by this instruction
Condition Code = 0010

6.5.9 Subtract Floating-Point (SE, SER)

Subtract Floating-Point (SE)
Subtract Floating-Point Register (SER)

Assembler Notation		Op-Code	Format
SE	R1,D2(X2)	6B	RX1,RX2
SE	R1,A2(FX2,SX2)	6B	RX3
SER	R1,R2	2B	RR

Operation

The two operand exponents are compared. If the exponents differ, the fraction with the smaller exponent is shifted right hexadecimally (four bits at a time), and its exponent is incremented by one for each hexadecimal shift, until the two exponents are equal. The hexadecimal digits (of four bits each) are shifted through the guard digits for additional precision. If no equalizing shifts are required, the guard digits remain zero. The second operand fraction is then subtracted algebraically from the first operand fraction. The guard digits participate in this subtraction.

If the subtraction of fractions produces a carry, the exponent of the result is incremented by one, and the fraction of the result is shifted right one hexadecimal digit. The carry bit is shifted back into the most significant hexadecimal digit of the fraction, producing a normalized result. This result is then R*-rounded and replaces the contents of the single-precision floating-point register specified by R1.

If the subtraction of fractions does not produce a carry, the result is normalized, if necessary, then R*-rounded. This result replaces the contents of the single-precision floating-point register specified by R1.

Condition Code

C	V	G	L	
0	0	0	0	Floating-point result is zero
0	0	0	1	Floating-point result is less than zero
0	0	1	0	Floating-point result is greater than zero
0	1	0	1	Exponent overflow, result is less than zero
0	1	1	0	Exponent cverflow, result is greater than zero
0	1	0	0	Exponent underflow

Programming Notes

When the subtraction of the fractions produces a carry, incrementing the exponent of the result by one can produce exponent overflow. In this case, the arithmetic fault interrupt is taken, and the contents of R1 remain unchanged.

Normalization of the result can produce exponent underflow. If PSW bit 19 is set, an arithmetic fault interrupt is taken, and the register specified by R1 is unchanged. If exponent underflow occurs and bit 19 of the current PSW is zero, no arithmetic fault occurs. Zeros replace the contents of the register specified by R1.

In the RX formats, the second operand must be located on a fullword boundary.

Fastest results occur when the first operand is larger than the second operand.

Example: SE

This example subtracts the contents of LOC from the contents of floating-point register 8 and places the result in floating-point register 8.

Floating-point REG8 contains X'7EFF FFFF'
LOC contains X'7A10 0000'

Assembler Notation	Comments
SE REG8,LOC	SUBTRACT (LOC) FROM (REG8)

Result of SE Instruction

(Floating-point REG8) = 7EFF FFEF
(LOC) unchanged by this instruction
Condition Code = 0010

6.5.10 Compare Floating-Point (CE, CER)

Compare Floating-Point (CE)
Compare Floating-Point Register (CER)

Assembler Notation		Op-Code	Format
CE	R1,D2(X2)	69	RX1,RX2
CE	R1,D2(FX2,SX2)	69	RX3
CER	R1,R2	29	RR

Operation

The first and second operands are compared. Comparison is algebraic, and the sign, fraction, and exponent of each number must be considered. The result is indicated by the condition code setting. Neither operand is changed.

Condition Code

C	V	G	L	
0	X	0	0	First operand is equal to second operand
1	X	0	1	First operand is less than second operand
0	X	1	0	First operand is greater than second operand

Programming Notes

The state of the V flag is undefined.

In the RX formats, the second operand must be located on a fullword boundary.

6.5.11 Multiply Floating-Point (ME, MER)

Multiply Floating-Point (ME)
Multiply Floating-Point Register (MER)

Assembler Notation		Op-Code	Format
ME	R1,D2(X2)	6C	RX1,RX2
ME	R1,A2(FX2,SX2)	6C	RX3
MER	R1,R2	2C	RR

Operation

The exponents of each operand, as derived from the excess-64 notation used in floating-point representation, are added to produce the exponent of the result. This exponent is converted back to excess-64 notation, and the fractions are then multiplied.

If the product is zero, the entire floating-point value is forced to zero, X'0000 0000'. If the product is not zero, the result is normalized. The sign of the result is determined by the rules of algebra. The R*-rounded result replaces the contents of the single-precision floating-point register specified by R1.

Condition Code

C	V	G	L	
0	0	0	0	Floating-point result is zero
0	0	0	1	Floating-point result is less than zero
0	0	1	0	Floating-point result is greater than zero
0	1	0	1	Exponent overflow, result is less than zero
0	1	1	0	Exponent overflow, result is greater than zero
0	1	0	0	Exponent underflow

Programming Notes

Multiplication of two 6-hexadecimal-digit fractions effectively produces a result of six hexadecimal digits and a number of guard digits. The guard digits participate in the R*-rounding of the final result.

The addition of exponents can produce exponent overflow. In this case, an arithmetic fault interrupt is taken, and both operands remain unchanged.

The addition of exponents or the normalization process can produce exponent underflow. If PSW bit 19 is set, an arithmetic fault interrupt is taken and the register specified by R1 is unchanged. If exponent underflow occurs and bit 19 of the current PSW is zero, no arithmetic fault occurs. Zeros replace the contents of the register specified by R1.

In the RX formats, the second operand must be located on a fullword boundary.

Fastest results occur when the second operand multiplier contains sets of four or more contiguous ones or zeros.

Example: ME

This example multiplies the contents of floating-point register 8 by the contents of memory location LOC and places the result in floating-point register 8.

Floating-point REG8 contains X'5FFF FFFF'
LOC contains X'60FF FFFF'

Assembler Notation	Comments
ME REG8,LOC	MULTIPLY (REG8) BY (LOC)

Result of ME Instruction

(Floating-point REG8) = 7FFF FFFE
(LOC) unchanged by this instruction
Condition Code = 0010

6.5.12 Divide Floating-Point (DE, DER)

Divide Floating-Point (DE)
Divide Floating-Point Register (DER)

Assembler Notation		Op-Code	Format
DE	R1,D2 (X2)	6D	RX1,RX2
DE	R1,A2 (FX2,SX2)	6D	RX3
DER	R1,R2	2D	RR

Operation

The exponents of each operand, as derived from the excess-64 notation used in floating-point representation, are subtracted to produce the exponent of the result. This exponent is converted back to excess-64 notation.

The first operand fraction is then divided by the second operand fraction. Division continues until the quotient is normalized, adjusting the exponent for each additional division required.

No remainder is returned. The sign of the quotient is determined by the rules of algebra. The R*-rounded quotient replaces the contents of the single-precision floating-point register specified by R1.

Condition Code

C	V	G	L	
0	0	0	0	Floating-point result is zero
0	0	0	1	Floating-point result is less than zero
0	0	1	0	Floating-point result is greater than zero
0	1	0	1	Exponent cverflow, result is less than zero
0	1	1	0	Exponent cverflow, result is greater than zero
0	1	0	0	Exponent underflow
1	1	0	0	Divisor equal to zero

Programming Notes

Before starting the divide operation, the divisor is checked. If it is equal to zero, the operation is aborted, and the arithmetic fault interrupt is taken. Neither operand is changed.

Subtraction of exponents may produce exponent overflow. In this case, an arithmetic fault interrupt is taken, and both operands remain unchanged.

The subtraction of exponents or the division process can produce exponent underflow; normalization of the result can produce exponent underflow. If PSW bit 19 is set, an arithmetic fault interrupt is taken, and the register specified by R1 is unchanged. If exponent underflow occurs and bit 19 of the current PSW is zero, no arithmetic fault occurs. Zeros replace the contents of the register specified by R1.

The 6-hexadecimal digit first operand fraction is divided by the 6-hexadecimal digit second operand, effectively producing the 6-hexadecimal digit quotient along with a number of guard digits. The guard digits participate in the R*-rounding of the final result.

In the RX formats, the second operand must be located on a fullword boundary.

Example: DE

This example divides the contents of floating-point register 4 by the contents of memory location LOC and places the result in floating-point register 4.

Floating-point REG4 contains X'44FF FFFF' = dividend
LOC contains X'0611 1111' = divisor

Assembler Notation	Comments
DE REG4,LOC	DIVIDE (REG4) BY (LOC)

Result of DE Instruction:

(Floating-point REG4) = 7FF0 0000
(LOC) unchanged by this instruction
Condition Code = 0010

6.5.13 Fix Register (FXR)

Assembler Notation	Op-Code	Format
FXR R1,R2	2E	RR

Operation

R1 and R2 specify a general-purpose register and a floating-point register respectively. The normalized floating-point number contained in the floating-point register is converted to a two's complement notation integer value by shifting and truncating. The result is stored in the general register specified by R1.

Condition Code

C	V	G	L	
X	0	0	0	Result is zero or underflow
X	0	0	1	Result is less than zero
X	0	1	0	Result is greater than zero
X	1	0	1	Overflow, result is less than zero
X	1	1	0	Overflow, result is greater than zero

Programming Notes

The range of floating-point magnitudes (M) that produces a nonzero integral result is:

$$+X'4880\ 0000' \geq M \geq +X'4110\ 0000'$$

Floating-point magnitudes greater than +X'487F FFFF' or -X'4880 0000' cause overflow. The result is forced to X'7FFF FFFF' if positive, or to X'8000 0000' if negative. The V flag is set in the condition code along with either the G or L flag, depending on the sign of the result.

Floating-point magnitudes less than +X'41100000' cause underflow, and the result is forced to zero.

In the event of overflow or underflow, no arithmetic fault interrupt is taken, even if enabled in the current PSW.

Example: FXR

This example converts the contents of floating-point register 8 to a fixed-point number and places it in register 3.

Floating-point REG8 contains X'46FF FF00'
REG3 contains unknown data

Assembler Notation	Comments
FXR REG3,REG8	CONVERT (REG8) TO FIXED-POINT

Result of FXR Instruction

(REG3) = 00FFFF00
(Floating-point REG8) unchanged by this instruction
Condition Code = 0010

6.5.14 Float Register (FLR)

Assembler Notation	Op-Code	Format
FLR R1,R2	2F	RR

Operation

R1 and R2 specify a floating-point register and a general-purpose register, respectively. The integer value contained in the general register specified by R2 is converted to a floating-point number and stored in the single-precision floating-point register specified by R1.

Condition Code

C	V	G	L	
X	0	0	0	Floating-point result is zero
X	0	0	1	Floating-point result is less than zero
X	0	1	0	Floating-point result is greater than zero

Programming Note

The full range of fixed-point integer values can be converted to floating-point. The fixed-point value X'7FFF FFFF', the largest positive integer, converts to the floating-point value X'487F FFFF'. The fixed-point value X'8000 0000', the most negative integer, converts to the floating-point value X'C880 0000'. The result in R1 is normalized and truncated, if necessary, to fit in the six fraction digits.

Example: FLR

This example converts the fixed-point contents of Register 4 to a floating-point number and places it in floating-point register 8.

REG4 contains X'7FFF FFF0'
Floating-point REG8 contains unknown data

Assembler Notation	Comments
FLR REG8,REG4	CONVERT (REG4) TO FLOATING-POINT

Result of FLR Instruction:

(Floating-point REG8) = 487FFFFF
(REG4) unchanged by this instruction
Condition Code = 0010

6.5.15 Load Double-Precision Floating-Point (LD, LDR, LDGR)

Load Double-Precision Floating-Point (LD)
Load Register Double-Precision Floating-Point (LDR)
Load Double-Precision Floating-Point Registers from General Registers (LDGR)

Assembler Notation		Op-Code	Format
LD	R1,D2(X2)	78	RX1,RX2
LD	R1,A2(FX2,SX2)	78	RX3
LDR	R1,R2	38	RR
LDGR	R1,R2	A6	RR

Operation

The floating-point second operand is normalized, if necessary, and placed in the double-precision floating-point register specified by R1.

Condition Code

C	V	G	L	
0	0	0	0	Double-precision result is zero
0	0	0	1	Double-precision result is less than zero
0	0	1	0	Double-precision result is greater than zero
0	1	0	0	Exponent underflow

Programming Notes

If the argument fraction is zero, the entire result is forced to zero, X'0000 0000 0000 0000'.

Normalization can produce exponent underflow. If PSW bit 19 is set, the arithmetic fault interrupt is taken, and the register specified by R1 remains unchanged. If exponent underflow occurs, and bit 19 of the current PSW is zero, no arithmetic fault occurs. Zeros replace the contents of the register specified by R1.

In the RX formats, the second operand must be located on a fullword boundary.

The R2 field for LDGR must specify the even member of an even/odd pair of general registers. The register specified by R2 contains the most significant 32 bits, and R2+1 contains the least significant 32 bits. If R2 is not an even numbered register, unpredictable results occur.

6.5.16 Load Positive Double-Precision Register (LPDR)

Assembler Notation	Op-Code	Format
LPDR R1,R2	33	RR

Operation

The double-precision floating-point second operand contained in the double-precision floating-point register specified by R2 is forced positive. The result is normalized if necessary and placed in the double-precision floating-point register specified by R1.

Condition Code

C	V	G	L	
0	0	0	0	Double-precision result is zero
0	0	1	0	Double-precision result is greater than zero
0	1	0	0	Exponent underflow

Programming Notes

If the argument fraction is zero, the entire result is forced to zero, X'0000 0000 0000 0000'.

Normalization of the result can produce exponent underflow. If PSW bit 19 is set, the arithmetic fault interrupt is taken, and the register specified by R1 remains unchanged. If exponent underflow occurs, and bit 19 of the current PSW is zero, no arithmetic fault occurs. Zeros replace the contents of the register specified by R1.

6.5.17 Load Complement Double-Precision Register (LCDR)

Assembler Notation	Op-Code	Format
LCDR R1,R2	37	RR

Cperation

The sign of the double-precision floating-point second operand contained in the double-precision floating-point register specified by R2 is complemented. The result is normalized if necessary and placed in the double-precision floating-point register specified by R1.

Condition Code

C	V	G	L	
0	0	0	0	Double-precision result is zero
0	0	0	1	Double-precision result is less than zero
0	0	1	0	Double-precision result is greater than zero
0	1	0	0	Exponent underflow

Programming Notes

If the argument fraction is zero, the entire result is forced to zero, X'0000 0000 0000 0000'.

Normalization may produce exponent underflow. If PSW bit 19 is set, the arithmetic fault interrupt is taken and the register specified by R1 remains unchanged. If an exponent underflow occurs and bit 19 of the current PSW is zero, no arithmetic fault occurs. Zeros replace the contents of the register specified by R1 in this case.

6.5.18 Load Multiple Double-Precision Floating-Point (LMD)

Assembler Notation	Op-Code	Format
LMD R1,D2(X2)	7F	RX1,RX2
LMD R1,A2(FX2,SX2)	7F	RX3

Operation

Successive double-precision floating-point registers, starting with the register specified by R1, are loaded from successive fullword memory location pairs, starting with the address of the second operand. The process stops when double-precision floating-point register 14 has been loaded.

Condition Code

Unchanged

Programming Notes

Values loaded into the double-precision floating-point registers are assumed to be normalized, and no test or adjustment is performed.

The second operand must be located on a fullword boundary.

6.5.19 Load General Registers from Double-Precision Floating-Point Register (LGDR)

Assembler Notation	Op-Code	Format
LGDR R1,R2	16	RR

Operation

The double-precision floating-point second operand, contained in the double-precision register specified by R2, is placed in the general register pair specified by R1. The second operand is unchanged.

Condition Code

C	V	G	L	
0	0	0	0	Result is zero
0	0	0	1	Result is less than zero
0	0	1	0	Result is greater than zero

Programming Notes

The R1 field must specify the even member of the even/odd pair of general registers receiving the result. The even numbered register receives the most significant 32 bits while the next sequential odd numbered register receives the least significant 32 bits.

If R1 and R2 do not specify even numbered registers, unpredictable results occur.

6.5.20 Store Double-Precision Floating-Point (STD)

Assembler Notation	Op-Code	Format
STD R1,D2(X2)	70	RX1,RX2
STD R1,A2(FX2,SX2)	70	RX3

Operation

The floating-point first operand, contained in the double-precision floating-point register specified by R1, is placed in the double word memory location specified by the second operand address. The first operand is unchanged.

Condition Code

Unchanged

Programming Note

The second operand must be located on a fullword boundary.

6.5.21 Store Multiple Double-Precision Floating-Point (STMD)

Assembler Notation	Op-Code	Format
STMD R1,D2(X2)	7E	RX1,RX2
STMD R1,A2(FX2,SX2)	7E	RX3

Operation

The contents of successive double-precision floating-point registers, starting with the even numbered register specified by R1, are stored in successive fullword memory location pairs, starting with the address of the second operand. The operation stops when the contents of double-precision floating-point register 14 have been stored.

Condition Code

Unchanged

Programming Note

The second operand must be located on a fullword boundary.

6.5.22 Add Double-Precision Floating-Point (AD, ADR)

Add Double-Precision Floating-Point (AD)
Add Register Double-Precisicn Floating-Point (ADR)

Assembler Notation		Op-Code	Format
AD	R1,D2(X2)	7A	RX1,RX2
AD	R1,A2(FX2,SX2)	7A	RX3
ADR	R1,R2	3A	RR

Operation

The two operand exponents are compared. If the exponents differ, the fraction with the smaller exponent is shifted right hexadecimally (four bits at a time), and its exponent is incremented by one for each hexadecimal shift until the two exponents are equal. Hexadecimal digits are shifted through the guard digits to retain precision. The fractions are then added algebraically.

If the addition of fractions produces a carry, the exponent of the result is incremented by one and the fraction of the result is shifted right one hexadecimal position. The carry bit is shifted back into the most significant hexadecimal digit of the fraction, producing a normalized result. This result is R*-rounded and replaces the contents of the double-precision floating-point register specified by R1.

If the addition of fractions does not produce a carry, the result is normalized, if necessary, and placed in the double-precision floating-point register specified by R1.

Condition Code

C	V	G	L	
0	0	0	0	Double-precision result is zero
0	0	0	1	Double-precision result is less than zero
0	0	1	0	Double-precision result is greater than zero
0	1	0	1	Exponent cverflow, result is less than zero
0	1	1	0	Exponent cverflow, result is greater than zero
0	1	0	0	Exponent underflow

Programming Notes

When the addition of fractions produces a carry, incrementing the exponent of the result by one may produce exponent overflow. In this case, the arithmetic fault interrupt is taken and both operands remain unchanged.

Normalization of the result can produce exponent underflow. If PSW bit 19 is set, an arithmetic fault interrupt is taken, and the register specified by R1 is unchanged. If exponent underflow occurs and bit 19 of the current PSW is zero, no arithmetic fault occurs. Zeros replace the contents of the register specified by R1.

Fastest results occur when the first operand is larger than the second operand.

In the RX formats, the second operand must be located on a fullword boundary.

6.5.23 Subtract Double-Precision Floating-Point (SD, SDR)

Subtract Double-Precision Floating-Point (SD)
Subtract Register Double-Precision Floating-Point (SDR)

Assembler Notation		Op-Code	Format
SD	R1,D2(X2)	7B	RX1,RX2
SD	R1,A2(FX2,SX2)	7B	RX3
SDR	R1,R2	3B	RR

Operation

The two operand exponents are compared. If the exponents differ, the fraction with the smaller exponent is shifted right hexadecimally (four bits at a time), and its exponent is incremented by one for each hexadecimal shift, until the two exponents are equal. Hexadecimal digits are shifted through the guard digits to retain precision. The second operand fraction is then subtracted algebraically from the first operand fraction.

If the subtraction of fractions produces a carry, the exponent of the result is incremented by one and the fraction of the result is shifted right one hexadecimal position. The carry bit is shifted back into the most significant hexadecimal digit of the fraction producing a normalized result. This result is R*-rounded and replaces the contents of the double-precision floating-point register specified by R1.

If the subtraction of fractions does not produce a carry, the result is normalized, if necessary, then R*-rounded and placed in the double-precision floating-point register specified by R1.

Condition Code

C	V	G	L	
0	0	0	0	Double-precision result is zero
0	0	0	1	Double-precision result is less than zero
0	0	1	0	Double-precision result is greater than zero
0	1	0	1	Exponent overflow, result is less than zero
0	1	1	0	Exponent overflow, result is greater than zero
0	1	0	0	Exponent underflow

Programming Notes

When the subtraction of fractions produces a carry, incrementing the exponent of the result by one may produce exponent overflow. In this case, the arithmetic fault interrupt is taken and the contents of R1 remain unchanged.

Normalization of the result can produce exponent underflow. If PSW bit 19 is set, an arithmetic fault interrupt is taken, and the register specified by R1 is unchanged. If exponent underflow occurs and bit 19 of the current PSW is zero, no arithmetic fault occurs. Zeros replace the contents of the register specified by R1.

Fastest results occur when the first operand is larger than the second operand.

In the RX formats, the second operand must be located on a fullword boundary.

6.5.24 Compare Double-Precision Floating-Point (CD, CDR)

Compare Double-Precision Floating-Point (CD)
Compare Register Double-Precision Floating-Point (CDR)

Assembler Notation		Op-Code	Format
CD	R1,D2(X2)	79	RX1,RX2
CD	R1,A2(FX2,SX2)	79	RX3
CDR	R1,R2	39	RR

Operation

The first and second operands are compared. Comparison is algebraic, taking into account the sign, exponent and fraction of each number. The result is indicated by the condition code setting. Neither operand is changed.

Condition Code

C	V	G	L	
0	X	0	0	First operand is equal to second operand
1	X	0	1	First operand is less than second operand
0	X	1	0	First operand is greater than second operand

Programming Notes

The state of the overflow flag is undefined.

In the RX formats, the second operand must be located on a fullword boundary.

6.5.25 Multiply Double-Precision Floating-Point (MD, MDR)

Multiply Double-Precision Floating-Point (MD)
Multiply Register Double-Precision Floating-Point (MDR)

Assembler Notation		Op-Code	Format
MD	R1,D2(X2)	7C	RX1,RX2
MD	R1,A2(FX2,SX2)	7C	RX3
MDR	R1,R2	3C	RR

Operation

The exponents of the two operands, as derived from the excess-64 notation used in floating-point representation, are added to produce the exponent of the result. This exponent is converted back to excess-64 notation. The fractions are then multiplied.

If the product is zero, the entire double-precision value is forced to zero, X'0000 0000 0000 0000'. If the product is not zero, the result is normalized, if necessary. The sign of the result is determined by the rules of algebra. The R*-rounded result replaces the contents of the double-precision floating-point register specified by R1.

Condition Code

C	V	G	L	
0	0	0	0	Double-precision result is zero
0	0	0	1	Double-precision result is less than zero
0	0	1	0	Double-precision result is greater than zero
0	1	0	1	Exponent overflow, result is less than zero
0	1	1	0	Exponent overflow, result is greater than zero
0	1	0	0	Exponent underflow

Programming Notes

Multiplication of two 14-hexadecimal-digit fractions effectively produces a result of 14 hexadecimal digits and a number of guard digits. The guard digits participate in the R*-rounding of the final result.

The addition of exponents may produce exponent overflow. In this case, an arithmetic fault interrupt is taken and both operands remain unchanged.

Normalization of the result can produce exponent underflow. If PSW bit 19 is set, an arithmetic fault interrupt is taken, and the register specified by R1 is unchanged. If exponent underflow occurs and bit 19 of the current PSW is zero, no arithmetic fault occurs. Zeros replace the contents of the register specified by R1.

In the RX formats, the second operand must be located on a fullword boundary.

Fastest results occur when the second operand multiplier contains sets of 4 or more contiguous cnes or zeros.

6.5.26 Divide Double-Precisicn Floating-Point (DD, DDR)

Divide Double-Precision Floating-Point (DD)
Divide Register Double-Precision Floating-Point (DDR)

Assembler Notation		Op-Code	Format
DD	R1,D2(X2)	7D	RX1,RX2
DD	R1,A2(FX2,SX2)	7D	RX3
DDR	R1,R2	3D	RR

Operation

The exponents of the two operands, as derived from the excess-64 notation used in floating-point representation, are subtracted to produce the exponent of the result. This exponent is converted back to excess-64 notation.

The first operand fraction is then divided by the second operand fraction. Division continues until the quotient is normalized, adjusting the exponent for each additional division required.

No remainder is returned. The sign of the result is determined by the rules of algebra. The R*-rounded quotient replaces the contents of the double-precision floating-point register specified by R1.

Condition Code

C	V	G	L	
0	0	0	0	Double-precision result is zero
0	0	0	1	Double-precision result is less than zero
0	0	1	0	Double-precision result is greater than zero
0	1	0	1	Exponent cverflow, result is less than zero
0	1	1	0	Exponent overflow, result is greater than zero
0	1	0	0	Exponent underflow
1	1	0	0	Divisor equal to zero

Programming Notes

Before starting the divide operation, the divisor is checked. If it is equal to zero, the operation is aborted, and the arithmetic fault interrupt is taken. Neither operand is changed.

The subtraction of exponents may produce exponent overflow. In this case, an arithmetic fault interrupt is taken and both operands remain unchanged.

Subtraction of exponents or the division process can produce exponent underflow. Normalization of the result can produce exponent underflow. If PSW bit 19 is set, an arithmetic fault interrupt is taken, and the register specified by R1 is unchanged. If exponent underflow occurs and bit 19 of the current PSW is zero, no arithmetic fault occurs. Zeros replace the contents of the register specified by R1.

The 14-hexadecimal-digit first operand fraction is divided by the 14-hexadecimal-digit second operand fraction, effectively producing the 14-hexadecimal-digit quotient along with a number of guard digits. The guard digits participate in the R*-rounding of the final result.

In the RX formats, the second operand must be located on a fullword boundary.

6.5.27 Fix Register Double-Precision (FXDR)

Assembler Notation	Op-Code	Format
FXDR R1,R2	3E	RR

Operation

R1 and R2 specify a general purpose register and a double-precision floating-pcint register, respectively. The normalized floating-point number contained in the floating-point register is converted to an integer value by shifting and truncating. The result is placed in the general register specified by R1.

Condition Code

C	V	G	L	
X	0	0	0	Result is zero or underflow
X	0	0	1	Result is less than zero
X	0	1	0	Result is greater than zero
X	1	0	1	Overflow, result is less than zero
X	1	1	0	Overflow, result is greater than zero

Programming Notes

The range of the floating-point magnitude (M) that produces a nonzero integral result is:

$$+ X'4880\ 0000\ 0000\ 0000' \geq M \geq + X'4110\ 0000\ 0000\ 0000'$$

Double-precision floating-point magnitudes greater than +X'487F FFFF FFFF FFFF' or -X'4880 0000 0000 0000' cause overflow. The result is forced to X'7FFF FFFF' if positive or to X'8000 0000' if negative. The V flag is set in the condition code along with either the G or L flag, depending on the sign of the result.

Double-precision floating-point magnitudes less than +X'4110 0000 0000' cause underflow, and the result is forced to zero.

In the event of overflow or underflow, no arithmetic fault interrupt is taken even if enabled in the current PSW.

6.5.28 Float Register Double-Precision (FLDR)

Assembler Notation	Op-Code	Format
FLDR R1,R2	3F	RR

Operation

R1 and R2 specify a double-precision floating-point register and a general purpose register, respectively. The integer value contained in the general register specified by R2 is converted to a floating-point number and placed in the double-precision floating-point register specified by R1.

Condition Code

C	V	G	L	
X	0	0	0	Double-precision result is zero
X	0	0	1	Double-precision result is less than zero
X	0	1	0	Double-precision result is greater than zero

Programming Notes

The full range of fixed-point integer values may be converted to double-precision floating-point. The fixed-point value X'7FFF FFFF', the largest positive integer, converts to a double-precision floating-point value of X'487F FFFF FF00 0000'. The fixed-point value X'8000 0000', the most negative integer, converts to a double-precision floating-point value of X'C880 0000 0000 0000'.

The result in R1 is normalized.

6.5.29 Load Single-Precision Floating-Point Register From Double (LED, LEDR)

Load Single-Precision Floating-Point Register from Double-Precision Memory (LED)
Load Single-Precision Floating-Point Register from Double-Precision Register (LEDR)

Assembler Notation		Op-Code	Format
LED	R1,D2(X2)	84	RX1,RX2
LED	R1,A2(FX2,SX2)	84	RX3
LEDR	R1,R2	A4	RR

Operation

Double-precision floating-point data from the second operand location is R*-rounded to single-precision accuracy, and placed in the single-precision floating-point register specified by R1.

Condition Code

C	V	G	L	
0	0	0	0	Floating-point result is zero
0	0	0	1	Floating-point result is less than zero
0	0	1	0	Floating-point result is greater than zero
0	1	0	0	Exponent underflow
0	1	0	1	Exponent overflow, result is less than zero
0	1	1	0	Exponent overflow, result is greater than zero

Programming Notes

R1 and R2 must specify even-numbered registers.

Rounding of the result may cause exponent overflow. In this case, the register specified by R1 is unchanged, and the arithmetic fault interrupt is taken.

Normalization of the result may produce exponent underflow. If enabled by PSW bit 19, the arithmetic fault interrupt is taken, and the register specified by R1 remains unchanged. If bit 19 of the current PSW is zero, zeros replace the contents of the register specified by R1.

In the RX formats, the second operand must be located on a fullword boundary.

6.5.30 Load Double-Precision Floating-Point Register From Single (LDE, LDER)

Load Double-Precision Floating-Point Register from Single-Precision Memory (LDE)
Load Double-Precision Floating-Point Register from Single-Precision Register (LDER)

Assembler Notation		Op-Code	Format
LDE	R1,D2(X2)	87	RX1,RX2
LDE	R1,A2(FX2,SX2)	87	RX3
LDER	R1,R2	A7	RR

Operation

Single-precision floating-point data from the second operand location is converted to double-precision data by appending trailing zeros. The result replaces the contents of the double-precision floating-point register specified by R1.

Condition Code

C	V	G	L	
0	0	0	0	Double-precision result is zero
0	0	0	1	Double-precision result is less than zero
0	0	1	0	Double-precision result is greater than zero
0	1	0	0	Exponent underflow

Programming Notes

The registers specified by R1 and R2 must be even-numbered registers.

Normalization of the result may produce exponent underflow. If enabled by PSW bit 19, the arithmetic fault interrupt is taken, and the register specified by R1 remains unchanged. If bit 19 of the current PSW is zero, no arithmetic fault occurs. Zeros replace the contents of the register specified by R1.

In the RX formats, the second operand must be located on a fullword boundary.

6.5.31 Store Double-Precision Floating-Point Register in Single-Precision Memory (STDE)

Assembler Notation	Op-Code	Format
STDE R1,D2(X2)	82	RX1,RX2
STDE R1,A2(FX2,SX2)	82	RX3

Operation

Data from the double-precision floating-point register specified by R1 is R*-rounded to single-precision accuracy, and stored in the fullword second operand lccation.

Condition Code

Unchanged

Programming Notes

The register specified by R1 must be an even-numbered register.

Normalization of the rounded result may produce exponent underflow. In this case, zero, X'0000 0000', replaces the contents of the second operand location.

Rounding of the result may cause exponent overflow. In this case, the contents of the second operand location remain unchanged, and the arithmetic fault interrupt is taken.

The second operand must be located on a fullword boundary.

CHAPTER 7
STRING OPERATIONS

7.1 INTRODUCTION

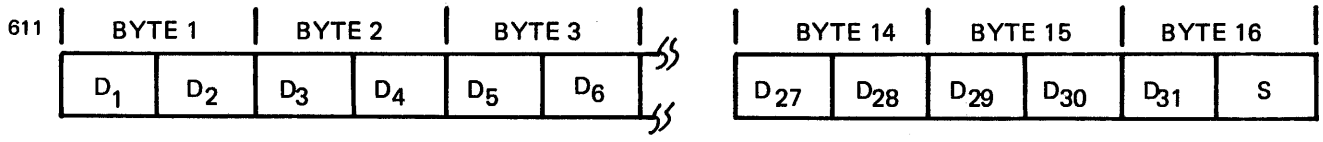
String operations deal with operands that are strings of consecutive bytes in memory beginning and ending on byte boundaries. Information contained in such a string may represent packed decimal data or ASCII character information including unpacked decimal data.

7.2 DECIMAL DATA FORMAT DEFINITIONS

Decimal operands can be in either packed or unpacked (zoned) format. The decimal operands are considered as right-aligned integers. The address of a decimal operand specifies the address of the left-most or most significant byte of the operand.

7.2.1 Packed Decimal

A number represented in packed decimal format is a fixed-point, signed integer, and consists of from 1 to 16 consecutive bytes. (See Figure 7-1.) Each byte is divided into two digit fields; thus each byte, except for the right-most in the string, contains two decimal digits represented in binary code. The only values allowed in a decimal digit field are 0 through 9. The right-most byte in the string contains the least significant decimal digit and the sign digit.



$D_1, D_2, D_3, \ldots\ldots D_{30}, D_{31}$ = DECIMAL DIGITS
S = SIGN DIGIT

Figure 7-1 Packed Decimal Format

There are two standard values for the sign S: hexadecimal C for plus and hexadecimal D for minus. However, the hexadecimal values 3, A, E, and F are also recognized for plus, and hexadecimal B is recognized for minus. Other values, 0 through 2 and 4 through 9, are illegal in the S position.

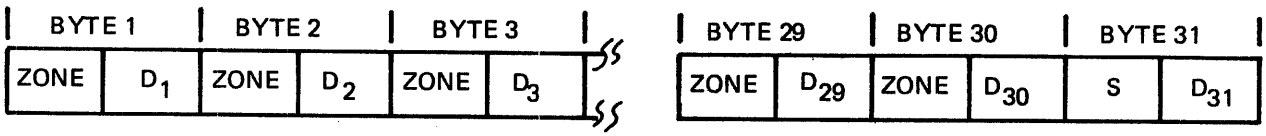
A packed decimal number contains an odd number of decimal digits. The most significant digit (zero or nonzero) of the number is in bit positions 0-3 of the left-most byte. The least significant digit occupies bit positions 0-3 of the right-most byte of the string, immediately preceding the sign digit, S. Any unused digit at the beginning of the string is filled with a leading zero.

7.2.2 Unpacked (Zoned) Decimal

A number represented in unpacked decimal format is a fixed-point signed integer, and consists of from 1 to 31 consecutive bytes. (See Figure 7-2.) Each byte, with the exception of the right-most byte, is assumed tc contain the 7-bit ASCII equivalent of a decimal digit. Thus, the top four bits contain zone information and the bottom four bits in each byte contain the binary equivalent of a decimal digit from 0 through 9.

When the processor generates an unpacked decimal byte string, the zone digit is always '3'. However, any zone value is accepted in an unpacked decimal operand, since the zone has no effect on the operation of the instructicns and is not examined. In the right-most byte of the string, S is the sign digit. Acceptable values for the sign digit are the same as those defined for packed decimal data.

612



ZONE = ZONE DIGIT
$D_1, D_2, D_3 \ldots\ldots, D_{30}, D_{31}$ = DECIMAL DIGITS
S = SIGN DIGIT

Figure 7-2 Unpacked Decimal Format

The most significant digit of an unpacked decimal number occupies the left-most byte of the string. The least significant digit occupies the right-most byte cf the string.

7.3 DECIMAL AND ALPHANUMERIC STRING INSTRUCTION FORMATS

The two binary/decimal conversion instructions use the standard RX format. The remaining string operations use the RXRX format.

In the instruction descriptions, the RXRX format is diagrammed as follows:

```
        {R1  }  {D2 (X2)      }  {R1  }  {D2 (X2)      }
OP      {=L1}, {A2 (FX2,SX2)} , {=L2}, {A2 (FX2,SX2)}
```

where any field may have either one of the options shown in the braces. R1/=L1 refers to the first operand length and R2/=L2 refers to the second operand length. Length of operand strings is always expressed as a number of bytes. These can vary from 0 to 15 for immediate length formats, and from 0 to maximum memory for register length. See Section 1.8.11 for further details of the RXRX instruction format.

7.4 STRING INSTRUCTIONS

The string instructions are interruptible, and use the scratchpad registers. If an interrupt occurs during the execution of a string instruction, bit 14 (IIP) is set by the processor in the old PSW to indicate that the scratchpad registers contain information pertinent to the interrupted instruction. See Section 10.3.4 for further information.

The instructions described in this section are:

LPB	Load Packed Decimal String as Binary (convert from decimal to binary)
STBP	Store Binary as Packed Decimal String (convert from binary tc decimal)
MVTU	Move Translated Until
MOVE	Move and Pad
MOVEP	Move and Pad with Default Pad
CPAN	Compare Alphanumeric
CPANP	Compare Alphanumeric with Default Pad
PMV	Pack and Move (convert unpacked decimal string to packed decimal string)
PMVA	Pack and Move Absolute (force positive result)
UMV	Unpack and Move (convert packed decimal string to unpacked decimal string)
UMVA	Unpack and Move Absolute (force positive result)

7.4.1 Load Packed Decimal String as Binary (LPB)

Assembler Notation	Op-Code	Format
LPB R1,D2(X2)	6F	RX1,RX2
LPB R1,A2(FX2,SX2)	6F	RX3

Operation

The second operand address points to the left-most byte of a packed decimal string of length sixteen bytes (31 packed decimal digits plus sign). Digits of the operand are checked for validity as the operand is converted to a 64-bit, two's complement binary number. The result replaces the contents of the even/odd general register pair specified by R1 and R1+1.

Condition Code

C	V	G	L	
0	0	0	0	Result is zero
0	0	0	1	Result is less than zero
0	0	1	0	Result is greater than zero
0	1	0	0	Overflow

Programming Notes

This instruction is interruptible.

R1 must specify an even-numbered register. If not, unpredictable results occur.

If an illegal decimal digit or sign digit is detected during conversion, the registers specified by R1 and R1+1 remain unchanged, and a data format fault interrupt is taken.

The largest positive number that can be processed without overflow is 9,223,372,036,854,775,807.

7.4.2 Store Binary As Packed Decimal String (STBP)

Assembler Notation	Op-Code	Format
STBP R1,D2(X2)	6E	RX1,RX2
STBP R1,A2(FX2, SX2)	6E	RX3

Operation

The contents of the even/odd general register pair specified by R1 and R1+1 are converted and stored in memory as a packed decimal string of length 16 bytes (31 packed decimal digits plus sign). The left-most byte is stored at the address specified by the second operand.

Condition Code

C	V	G	L	
0	0	0	0	Result is zero
0	0	0	1	Result is less than zero
0	0	1	0	Result is greater than zero

Programming Notes

This instruction is interruptible.

R1 must specify an even-numbered register. If not, unpredictable results occur.

7.4.3 Move Translated Until (MVTU)

Assembler Notation	Op-Code	Function Code	Format
MVTU {R1 / =L1},{D2(X2) / A2(FX2,SX2)},{R2 / =L2},{D2(X2) / A2(FX2,SX2)}	8C	00	RXRX

Operation

General register 0 contains the escape character whose occurrence causes the instruction to terminate. General register 2 contains the address of a translation table. This translation table is a simple list of 256 single byte entries, not to be confused with the table used by the translate instruction. The first operand string begins at the address specified by the first operand address. The length of this string is equal to either the contents of the register specified by R1, or the value of L1. The second operand string begins at the address specified by the second operand address. The length of this string is equal to either the contents of the register specified by R1, or the value of L2.

Successive bytes from the second operand string are moved to the first operand string, as follows:

1. A byte is fetched from the second operand string (this is the argument byte). The contents of general register 2 are tested. If general register 2 contains zero, no translation occurs. If general register 2 does not contain zero, it contains the address of a translation table of maximum size 256 bytes. In this case, the argument byte fetched from the second operand string is used as an index into the translation table, and the byte at the resulting address is fetched and used as the argument byte.

2. The argument byte is compared with the escape character contained in bits 24:31 of general register 0. If the bytes are the same, the C flag is set in the condition code, and the instruction terminates. Otherwise, the argument byte is stored in the first operand string, and the next successive byte is processed. This operation is repeated until either the escape character is encountered, the first operand string has been filled, or the second operand string has been exhausted.

3. When the instruction terminates, the address of the next byte to be moved from the second operand string is returned in general register 1.

Condition Code

C	V	G	L	
0	0	0	0	Entire string moved
0	1	0	0	First operand filled before entire string moved
1	0	0	0	Escape character encountered

Programming Notes

This instruction is interruptible.

The contents of general register 1 may change during instruction execution, but are not valid until instruction termination.

Bytes are moved from the second operand string to the first operand string in a left-to-right sequence. If the strings overlap, such that the source is to the left of the destination, unpredictable results occur.

7.4.4 Move (MOVE, MOVEP)

Move and Pad (MOVE)
Move and Pad with Default Pad (MOVEP)

	Assembler Notation	Op-Code	Function Code	Format
MOVE	{R1 / =L1},{D2(X2) / A2(FX2,SX2)},{R2 / =L2},{D2(X2) / A2(FX2,SX2)}	8C	01	RXRX
MOVEP	{R1 / =L1},{D2(X2) / A2(FX2,SX2)},{R2 / =L2},{D2(X2) / A2(FX2,SX2)}	8C	21	RXRX

Operation

The first operand string begins at the address specified by the first operand address and has a length equal either to the contents of the register specified by R1, or to the value of L1. The second operand string begins at the address specified by the second operand address and has a length equal either to the contents of the register specified by R2, or to the value of L2.

Successive bytes from the second operand string are moved to the first operand string. If the second operand string is exhausted before the first operand string is filled, the remaining bytes in the first operand string are filled using the pad character. If MOVE is specified, the pad character is contained in bits 24:31 of general register 0. If MOVEP is specified, the remainder of the first operand is filled with ASCII space characters (X'20'). If the first operand string is filled before the second operand string is exhausted, overflow results, and the operation is terminated.

When the instruction terminates, the address of the next byte to be moved from the second operand string is returned in general register 1.

Condition Code

C	V	G	L	
0	0	0	0	Entire string moved
0	1	0	0	First operand filled before entire string moved

Programming Notes

These instructions are interruptible.

The contents of general register 1 may change during instruction execution, but are not valid until instruction termination.

If MCVEP is specified, the contents of general register 0 are ignored.

Bytes are moved from the seccnd operand string to the first operand string in a left-to-right sequence. If the strings overlap such that the source is to the left of the destination, unpredictable results occur.

7.4.5 Compare (CPAN, CPANP)

Compare Alphanumeric (CPAN)
Compare Alphanumeric with Default Pad (CPANP)

Assembler Notation		Op-Code	Function Code	Format
CPAN	{ R1 / =L1 }, { D2(X2) / A2(FX2,SX2) }, { R2 / =L2 }, { D2(X2) / A2(FX2,SX2) }	8C	02	RXRX
CPANP	{ R1 / =L1 }, { D2(X2) / A2(FX2,SX2) }, { R2 / =L2 }, { D2(X2) / A2(FX2,SX2) }	8C	22	RXRX

Operation

The first operand string begins at the address specified by the first operand address and has a length equal either to the contents of the register specified by R1, or to the value of L1. The second operand string begins at the address specified by the second operand address and has a length equal either to the contents of the register specified by R2, or to the value of L2.

The two strings are compared a byte at a time until the first unequal byte pair is found, or until the length of both strings is exhausted.

If the strings are of unequal length, the shorter string is logically extended to the length of the longer string. If CPAN is specified, this is done by using the pad character contained in bits 24:31 of general register 0. If CPANP is specified, the ASCII space character (X'20') is used as the default pad character.

Upon termination, general register 1 is set equal to one less than the number of second operand bytes that successfully matched corresponding bytes in the first operand string. This count (or offset) includes pad characters if the second operand string was longer than the first.

For example, a first operand string of length three bytes contains the characters ABC. A second operand string of length six bytes contains the characters ABCDDD.

A CPANP instruction returns a condition code of 0001 (first operand string less than second operand string) and general register 1 is set equal to 2. The first nonmatching character was the character 'D' in the second operand string. Given the same operand strings, a CPAN instruction with general register 0 set equal to a pad character of 'D' returns a condition code of 0000 (strings are equal including pad characters) and general register 1 is set equal to 5.

Condition Code

C	V	G	L	
0	0	0	0	Strings are equal
0	0	1	0	First operand string greater than second operand string
1	0	0	1	First operand string less than second operand string

Programming Notes

If CPANP is specified, the contents of general register 0 are ignored. If CPAN is specified, bits 0:23 of general register 0 are ignored.

These instructions are interruptible.

7.4.6 Pack and Move (PMV, PMVA)

Pack and Move (PMV)
Pack and Move Absolute (PMVA)

Assembler Notation		Op-Code	Function Code	Format
PMV	{ R1 | =L1 }, { D2(X2) | A2(FX2,SX2) }, { R2 | =L2 }, { D2(X2) | A2(FX2,SX2) }	8C	03	RXRX
PMVA	{ R1 | =L1 }, { D2(X2) | A2(FX2,SX2) }, { R2 | =L2 }, { D2(X2) | A2(FX2,SX2) }	8C	23	RXRX

Operation

The first operand string begins at the address specified by the first operand address. The length of this string in bytes is one greater than either the contents of the register specified by R1, or the value of L1. The second operand string begins at the address specified by the second operand address. The length of this string in bytes is one greater than either the contents of the register specified by R1, or the value of L2.

The second operand string consists of unpacked decimal data digits with a sign digit. Data in this string is packed and replaces the first operand string. Leading zeros are supplied as required to fill the higher-order positions of the first operand string.

Condition Code

C	V	G	L	
0	0	0	0	Result is zero
0	X	0	1	Result is less than zero
0	X	1	0	Result is greater than zero
0	1	X	X	Overflow
1	X	X	X	Invalid digit in second operand string

Programming Notes

PMVA causes the sign digit cf the first operand string to be forced positive.

Overflow occurs if the length of the first operand string is not sufficient to contain the packed representation of the second operand string. The V flag is set in the condition code, and the specified number of digits in the first operand string receive packed data from the second operand string. Higher-order digits of packed data are lost in this case.

Leading zero digits do not cause overflow. They are truncated if necessary.

These instructions are interruptible instructions.

Since packing is done conceptually from right to left with any overlapping allowed, the instruction PMV can be used to check the validity of decimal data.

If the destination string is to the left of the source string, such that the signed byte of the destination string is taken as data from the source string, the sign digit is found to be an illegal data digit, and the C flag is set at completion of the instruction.

7.4.7 Unpack and Move (UMV, UMVA)

Unpack and Move (UMV)
Unpack and Move Absolute (UMVA)

	Assembler Notation	Op-Code	Function Code	Format
UMV	{ R1 / =L1 }, { D2(X2) / A2(FX2,SX2) }, { R2 / =L2 }, { D2(X2) / A2(FX2,SX2) }	8C	04	RXRX
UMVA	{ R1 / =L1 }, { D2(X2) / A2(FX2,SX2) }, { R2 / =L2 }, { D2(X2) / A2(FX2,SX2) }	8C	24	RXRX

Operation

The first operand string begins at the address specified by the first operand address. The length of this string in bytes is one greater than either the contents of the register specified by R1, or the value of L1. The second operand string begins at the address specified by the second operand address. The length of this string in bytes is one greater than either the contents of the register specified by R2, or the value of L2.

The second operand string consists of packed decimal data digits with a sign digit. Data in this string is unpacked and replaces the first operand string. Leading zeros are supplied as required to fill the higher-order positions of the first operand string.

Condition Code

C	V	G	L	
0	0	0	0	Result is zero
0	X	0	1	Result is less than zero
0	X	1	0	Result is greater than zero
0	1	X	X	Overflow
1	X	X	X	Invalid digit in second operand string

Programming Notes

UMVA causes the sign digit of the first operand string to be forced positive.

Overflow occurs if the length of the first operand string is not sufficient to contain the unpacked representation of the second operand string. The V flag is set in the condition code, and the specified number of digits in the first operand string receive unpacked data from the second operand string. Higher-order digits of unpacked data are lost in this case.

Leading zero digits do not cause overflow. They are truncated if necessary.

These instructions are interruptible instructions.

Since unpacking is done conceptually from right to left with any overlapping allowed, the instruction UMV can be used to check the validity of decimal data.

If the destination string is to the left of the source string, such that the signed byte of the destination string is taken as data from the source string, the sign digit is found to be an illegal data digit, and the C flag is set at the completion of the instruction.

CHAPTER 8
HIGH SPEED DATA HANDLING INSTRUCTIONS (OPTIONAL)

8.1 INTRODUCTION

The data handling instructions are used to compute polynomial error check redundancy characters, as used by most data communications protocols. Communications protocols supported by this option include, but are not limited to, the following:

1. Binary Synchronous Communications (BISYNC or BSC) - IBM's widely accepted half-duplex protocol uses the CRC BISYNC error check polynomial ($x^{16} + x^{15} + x^2 + 1$).

2. Synchronous Data Link Control (SDLC) - IBM's new full-duplex protocol uses the CRC SDLC error check polynomial ($x^{16} + x^{12} X^5 + 1$).

3. Advanced Data Communications Control Procedure (ADCCP) - ANSI's proposed National Standard full-duplex protocol uses CRC SDLC.

4. High Level Data Link Control (HDLC) - The International Standard Organizations full-duplex protocol uses CRC SDLC.

8.2 DATA HANDLING INSTRUCTION FORMATS

The optional data handling instructions use the Register to Register (RR), and the Register and Indexed Storage (RX) formats.

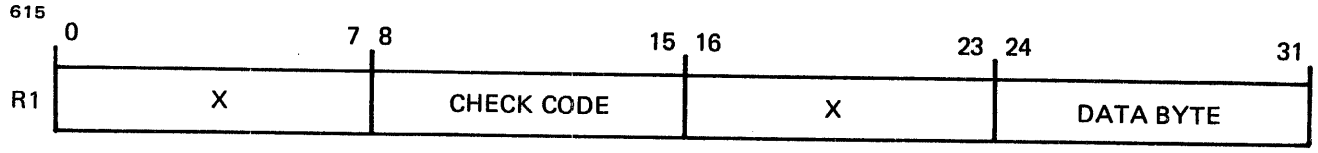
8.3 DATA HANDLING INSTRUCTIONS

PB	Process Byte
PBR	Process Byte Register

8.3.1 Process Byte (PB)

Assembler Notation	Op-Code	Format
PB R1,D2(X2)	62	RX1, RX2
PB R1,A2(FX2,SX2)	62	RX3

Set-Up



Bits 24:31 of the register specified by R1 contain the data byte to be processed. Bits 8:15 of the register specified by R1 contain a check code to indicate the type of processing. This byte is interpreted as follows:

X'00' Cumulative check zero (CRC BISYNC)
X'01' Cumulative check one (CRC SDLC)
X'02' Cumulative check two (LRC)

The second operand address points to a halfword residual checksum to be included in the cumulative check.

Operation

If CRC BISYNC is specified, the data byte and the old residual checksum participate in the generation of a new residual checksum based on the evaluation of the polynomial $(x^{16} + x^{15} + x^2 + 1)$.

If CRC SDLC is specified, a similar operation is performed, using the polynomial $(x^{16} + x^{12} + x^5 + 1)$.

In both of these cases, the new residual checksum replaces the old residual checksum at the second operand location.

If LRC is specified, the Exclusive-OR of the data byte with the old residual checksum replaces the old residual checksum at the second operand location.

Condition Code

Unchanged

Programming Notes

Bits 0:7 and 16:23 of the register specified by R1 are ignored.

The register specified by R1 remains unchanged.

The second operand must be located on a halfword boundary.

Undefined check codes should not be used. If they are, the results are undefined.

Example: PB

This example performs a Process Byte instruction and stores the residue in RESIDUE.

```
Register 1          contains X'C001007A'
    where:          01 = CRC SDLC
                    7A = DATA BYTE

RESIDUE             contains X'D053' = old residue
```

Assembler Notation	Comments
PB R1,RESIDUE	RESIDUE on halfword boundary

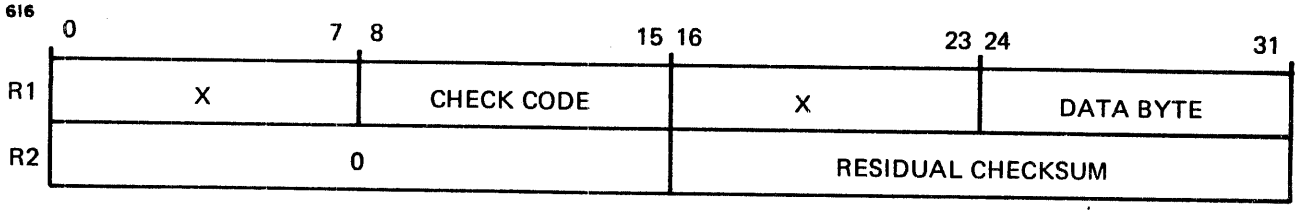
Result of PB Instruction

(R1) unchanged by this instruction
(RESIDUE) = X'BC13' = new residue
Condition Code unchanged by this instruction

8.3.2 Process Byte Register (PBR)

Assembler Notation	Op-Code	Format
PBR R1,R2	32	RR

Set-Up



Bits 24:31 of the register specified by R1 contain the data byte to be processed. Bits 8:15 of the register specified by R1 contain a check code indicating the type of processing. This byte is interpreted as follows:

X'00' Cumulative check zero (CRC BISYNC)
X'01' Cumulative check one (CRC SDLC)
X'02' Cumulative check two (LRC)

The second operand is a fullword contained in the register specified by R2. Bits 16:31 of the second operand contain the residual checksum to be included in the processing.

Operation

If CRC BISYNC is specified, the data byte and the old residual checksum participate in the generation of a new residual checksum, based on the evaluation of the polynomial ($x^{16} + x^{15} + x^2 + 1$).

If CRC SDLC is specified, a similar operation is performed, using the polynomial ($x^{16} + x^{12} + x^5 + 1$).

In both these cases, the new residual checksum replaces the contents of bits 16:31 of register specified by R2.

If LRC is specified, the Exclusive-OR of the data byte with the old residual checksum replaces the old residual checksum in the second operand.

Condition Code

Unchanged

Programming Notes

Bits 0:7 and 16:23 of the register specified by R1 are ignored. The register specified by R1 remains unchanged. Bits 0:15 of the register specified by R2 are not used and must be zero.

Undefined check codes should not be used. If they are, the results are undefined.

CHAPTER 9
INPUT/OUTPUT OPERATIONS

9.1 INTRODUCTION AND CONFIGURATION OF I/O SYSTEM

Input/Output (I/O) operations, as defined for the processor, provide a versatile means for the exchange of information between the processor, memory, and external devices. Communication between the processor and external devices is accomplished over the I/O bus. Data transfers over the I/O bus require processor intervention, either programmed or automatic, for each item transferred.

Direct data transfers between external devices and memory are accomplished over the DMA Bus, and proceed independently of the processor so other program prccessing can proceed simultaneously.

9.2 DEVICE CONTROLLERS

9.2.1 Function

The basic function of a device controller is:

1. To provide synchronization with the processor
2. To provide device address recognition
3. To transmit operational commands from the processor to the device
4. To translate device status into meaningful information for the processor
5. To request processor attention when required

In addition, a controller may generate parity; convert serial data to parallel; buffer incoming or outgoing data; or perform other device-dependent functicns.

9.2.2 Device Addressing

The system design allows as many as 1,023 external devices. Each device must have its own address or device number, ranging from X'001' through X'3FF'. (Device number X'000' is not assigned.) The minimum system provides for 255 device numbers. Larger systems may have either 511 or 1,023.

9.2.3 Processor/Controller Communication

Device controllers may communicate with the processor either directly, using the I/O bus, or indirectly through a selector channel. Communication between the processor and controller is a bidirectional, request/response operation.

The processor can initiate communication by sending the device number out onto the I/O bus. When a controller recognizes that number as its address, it returns a synchronization signal to the processor and remains ready to accept commands from the processor. The processor waits up to 40 microseconds for the synchronization signal. If no signal is received within this period, the processor aborts the operation and notifies the controlling program. In this case, the status returned is X'04' known as False Sync. The condition code in the PSW is also set to X'4' (V flag=1). Controller malfunction and software failure (incorrect device address) are the most common causes of this type of time-out.

A controller can initiate communication with the processor by generating an attention signal. If the processor is in an interruptible state as defined by bits 17 and 20 of the PSW, this signal causes the processor to temporarily suspend the normal "fetch instruction/execute/fetch next instruction" operation at the end of the execute phase, and to transmit an acknowledge signal over the I/O bus. The controller requesting attention responds with a synchronization signal and transmits its device number to the processor.

9.2.4 Device Priorities - External Interrupt Levels; Interrupt Queuing

External Interrupt Levels

The system architecture provides four external interrupt levels. PSW bits 17 and 20 define the external interrupt enable status of the processor.

When interrupt requests occur on more than one interrupt level, the request on the highest priority interrupt level is acknowledged first. Level 0 is the highest; level 3 is the lowest in priority.

Interrupt Queuing

Any device controller attempting to interrupt the processor activates one of the four attention lines sensed by the processor and holds that line active until the processor acknowledges the interrupt. Requests for attention are asynchronous; therefore more than one request may be pending at any time on any interrupt level. The system resolves these conflicts according to device priority, determined by the physical placement of the device controller on the I/O bus. When two or more device controllers on the same interrupt level request attention at the same time, the controller nearest to the processor in the RACK0/TACK0 priority wiring pattern captures the acknowledge signal from the processor and is serviced first. All other interrupting controllers of lower priority must wait for the next acknowledge signal from the processor.

9.3 INTERRUPT SERVICE POINTER TABLE

Device requests for service may result in either an immediate interrupt or an auto driver channel operation. The processor chooses one of these options according to information contained in the interrupt service pointer table.

The interrupt service pointer table is an ordered list containing one entry for each possible device number in the system. The table starts at memory location X'0000D0' and contains a halfword entry for each device number in the system. For a minimum system (255 device numbers), the table extends through memory location X'0002CF'; for a maximum system (1023 device numbers), the table extends through memory location X'0008CF'. The software controlling I/O operations must set up the table.

When the processor receives the device address after acknowledging a request for service, it adds twice the device address to X'0000D0'. The result is the address, within the table, of the entry reserved for the device requesting attention.

If the entry in the table is even (bit 15 equals 0), the processor takes an immediate interrupt and transfers control to the software interrupt service routine at the address contained in the table. If the entry in the table is odd (bit 15 equals 1), the processor transfers control to the auto driver channel, without interrupting the currently running program.

At the time the processor transfers control to the software interrupt service routine, the old PSW (current at the time of the device request) has been saved in registers 0 and 1 of the new register set. The device number is saved in register 2 and the status in register 3. The status portion of the current PSW has been replaced by the value X'000028nX', where n is the new register set number equal to the device interrupt level, and X is the least significant 4 bits of the device status. Machine malfunction interrupts and higher level I/O interrupts are enabled and all other interrupts are disabled. The entry in the interrupt service pointer table is now the new location counter.

9.4 CONTROL OF I/O OPERATIONS

The 32-bit I/O structure allows several data transfers depending on the particular application and on the characteristics of the external devices. Primary methods of data transfer between the processor and external devices are:

- One byte or one halfword to or from any of the general registers
- One byte or one halfword tc or from memory
- A block of data to or frcm memory under control of a selector channel or DMA universal interface
- Multiplexed blocks of data to or from memory under control of the auto driver channel

Standard device controllers require a predetermined sequence of commands to effect data transfers. These commands address the device, put it in the correct mode, and cause data to be transferred. Because all I/O instructions are privileged operations, I/O control programs must run in the supervisor mode, i.e., with bit 23 of the current PSW zero. I/O control programs should disable immediate interrupts or enable only higher level interrupts, as controlled by PSW bits 17 and 20.

9.5 STATUS MONITORING I/O

The simplest form of I/O programming is status monitoring I/O. In this mode of operation, only one device is handled at a time, and the processor cannot overlap other operations with the data transfer. The sequence of cperations in this type of programming is:

1. Address the device and set the proper mode (output command instruction).
2. Test the device status (sense status instruction).
3. Loop back to the sense status instruction until the status byte indicates that the device is ready (conditional branch instruction).
4. When the device is ready, transfer the data (read or write instruction).
5. If the transfer is not complete, branch back to the sense status instruction. If it is complete, terminate.

9.6 INTERRUPT DRIVEN I/O

Interrupt driven I/O allows the processor to take advantage of the disparity in speed between itself and the external devices being controlled. With status monitoring, the processor spends time waiting for the device. With interrupt driven programming, the processor can use this time performing other functions. This kind of programming establishes at least two levels of operation. On one level are the interrupt service programs. On the other level are interruptible programs that run with the immediate interrupt enabled.

Before starting interrupt driven operations, the interrupt service pointer table must be set up. This table starts at memory location X'0000D0' and must contain a halfword address entry for every possible device. The table is ordered according to device addresses in such a way that X'0000D0' plus two times the device address equals the memory address of the table entry reserved for that device. The value placed in the location reserved for a device is the address of the interrupt service routine for the device.

For example, if a terminal is connected at an address of X'02' and the interrupt routine resides in memory at address X'3000', the setup involves writing X'3000' at memory location X'D4'. Note that X'D4'=X'D0'+ 2 times the terminal address.

Although there may be gaps in device address assignments, the interrupt service pointer table should be completely filled. Entries for nonexistent devices should point to an error recovery routine. This precaution prevents system failure in the event of spurious interrupts caused by hardware malfunction or by improper use of the simulate interrupt instruction.

The next step is to prepare the device for the transfer, preferably with the immediate interrupts disabled. Once the table pointer has been set up and the device prepared, the processor can move on to an interruptible program.

The sequence of operation in this type of program is:

1. Set up the interrupt service pointer table to vector to error addresses for undefined devices.

2. Store the address of the software interrupt service routine at two times the device number plus X'D0' (X'D0' is starting address cf service pointer table).

3. Set up the software interrupt service routine.

4. Set up the device and enable device interrupts.

5. Enable I/O interrupts in the PSW.

When the device signals a need for service, the processor saves its current state and transfers control to the interrupt service routine at the location specified in the interrupt service pointer table. At this time, the current PSW has a status that indicates running state, machine malfunction interrupt enabled, higher level I/O interrupts enabled, and all other interrupts disabled. The condition code contains bits 4:7 of the device status. Registers 0 and 1 of the new set contain the old PSW, indicating the status and location of the interrupted program. Register 2 of that set contains the device address. Register 3 contains the device status.

The interrupt service routine should:

1. check the device status in Register 3, and if satisfactory,

2. make the transfer, and

3. return to the interrupted program by reloading the old PSW from registers 0 and 1 (LPSWR R0).

The interrupt service routine should not enable immediate interrupts on its own interrupt level. This would allow other interrupt requests to be acknowledged, and the contents of registers 0:4 could be lost. If it is necessary to enable immediate interrupts on the same level, the routine should save the register set, switch to a different register set, save it if necessary, and then enable immediate interrupts.

9.7 SELECTOR CHANNEL I/O

9.7.1 Introduction

The selector channel controls the transfer of data directly between high speed devices and memory. As many as 16 devices may be attached to the selector channel, only one of which may be operating at any one time. The advantage in using the selector channel is that other program processing may proceed simultaneously with the transfer of data between the external device and memory. This is possible because the selector channel accesses memory on a cycle stealing basis, permitting the processor and the channel to share memory. In some cases, execution time of the program in progress may be affected, while in others, the effect is negligible. This depends upon the rate at which the selector channel and processor compete for memory cycles.

The selector channel is linked to the processor over the I/O bus. It has its own unique device number which it recognizes when addressed by the processor. Like other device controllers, it can request processor attention through the immediate interrupt.

9.7.2 Selector Channel Devices

The selector channel has a private bus similar to the processor's I/O bus. Controllers for the devices associated with the selector channel are attached to this bus. When the selector channel is idle, its private bus is connected directly to the I/O bus. If this condition exists, the processor can address, command, and accept interrupt requests from the devices attached to the selector channel. When the selector channel is busy, this connection is broken. All communication between the processor and devices on the selector channel is cut off. Any attempt by the processor to address a device on the channel when the channel is busy results in instruction time-out.

9.7.3 Selector Channel Operation

Two registers in the selector channel hold the current memory address and the final memory address. With the use of write instructions, the control software places the address of the first byte of the data buffer into the current address register and the address of the last byte into the final address register. This is done before starting a selector channel operation. During the data transfer, the channel increments the current address register by one for each byte transferred. When the current address equals the final address, the last byte has been transferred, and the channel terminates.

The selector channel accesses memory a minimum of one halfword at a time; therefore, the transfer must always involve an integral number of halfwords. The starting address of the data buffer must always be on an even byte (halfword) boundary. The final address must always be on an odd byte boundary. The starting address must be less than the final address.

Upon termination, the software should read back from the selector channel the address contained in the current address register. If this address is not equal to the final address specified for the transfer, and if the buffer limits were properly checked before the transfer, this condition indicates a device malfunction or an unusual condition within the device. For example, crossing a cylinder boundary on a disc is an abnormal termination. The reason for the termination is indicated in the selector channel status or the device status.

9.7.4 Selector Channel Programming

The usual method of programming with the selector channel uses the immediate interrupt. The first step in the operation is to check the status of the selector channel. If the selector channel is not busy, the address of the termination interrupt service routine is placed in the location within the interrupt service pointer table reserved for the selector channel. The program should then proceed as follows:

1. Give the selector channel a command to stop. This command initializes the selector channel registers and assures the idle condition with the private bus connected to the I/O bus, so that the device may be set up for data transfer.

2. Give the selector channel the starting and final addresses.

3. Prepare the device for the transfer with the required commands and information.

4. Give the selector channel the command to start.

With the start command, the selector channel breaks the connection between its private bus and the processor's I/O bus, and provides a direct path between memory and the last device addressed over its bus. When the device becomes ready, the channel starts the transfer, which proceeds to completion without further processor intervention. Once the start command has been given, the processor can be directed to the execution of concurrent programs.

Upon termination, the channel signals the processor that it requires service. The processor subsequently takes an immediate interrupt, transferring control to the selector channel interrupt service routine. At this time, registers 0:3 of the new set are set up as for any other immediate interrupt.

If a power fail/restore sequence occurs while using the selector channel, the contents of the selector channel's internal registers are undefined.

9.8 I/O INSTRUCTION FORMATS

I/O instructions use the Register to Register (RR) and the Register and Indexed Storage (RX) instruction formats.

9.9 I/O INSTRUCTIONS

Following most I/O instructions, the V flag in the condition code indicates instruction time-out. This means that the operation was not completed, either because the device did not respond at all, or because it responded incorrectly.

In the Sense Status and Autoload instructions, the V flag can also mean examine status. To distinguish between these two conditions, the program should test bits 0:3 of the device status byte. If all of these bits are zero, device time-out has occurred.

The instructions described in this section are:

SS	Sense Status
SSR	Sense Status Register
OC	Output Command
CCR	Cutput Command Register
RD	Read Data
RDR	Read Data Register
RH	Read Halfword
RHR	Read Halfword Register
WD	Write Data
WDR	Write Data Register
WH	Write Halfword
WHR	Write Halfword Register
AL	Autoload
SCP	Simulate Channel Program

9.9.1 Output Command (OC, OCR)

Output Command (OC)
Output Command Register (OCR)

Assembler Notation		Op-Code	Format
OC	R1,D2(X2)	DE	RX1,RX2
OC	R1,A2(FX2,SX2)	DE	RX3
OCR	R1,R2	9E	RR

Operation

Bits 22:31 of the register specified by R1 contain the 10-bit device address. The processor addresses the device and transfers an 8-bit command byte from the second operand location to the device. Neither operand is changed.

Condition Code

C	V	G	L	
0	0	0	0	Operation successful
0	1	0	0	Instruction time-out (FALSE SYNC)

Programming Notes

In the RR format, bits 24:31 of the register specified by R2 contain the device command.

These instructions are privileged operations.

9.9.2 Sense Status (SS, SSR)

Sense Status (SS)
Sense Status Register (SSR)

Assembler Notation		Op-Code	Format
SS	R1,D2(X2)	DD	RX1,RX2
SS	R1,A2(FX2,SX2)	DD	RX3
SSR	R1,R2	9D	RR

Operation

Bits 22:31 of the register specified by R1 contain the 10-bit device address. The device is addressed and the 8-bit device status is transferred to the second operand location. The condition code is set equal to the least significant four bits of the device status byte. The first operand is unchanged.

Condition Code

Bits 4:7 of the device status byte are copied into the condition code. See the appropriate device manual for a description of this status.

If the device is not in the system, the condition code is set to 0100 (false sync). In this case, the status byte returned is X'04'.

Programming Notes

In the RR format, the device status byte replaces bits 24:31 of the register specified by R2. Bits 0:23 are forced to zero.

These instructions are privileged operations.

9.9.3 Read Data (RD, RDR)

Read Data (RD)
Read Data Register (RDR)

Assembler Notation		Op-Code	Format
RD	R1,D2(X2)	DB	RX1,RX2
RD	R1,A2(FX2,SX2)	DB	RX3
RDR	R1,R2	9B	RR

Operation

Bits 22:31 of the register specified by R1 contain the 10-bit device address. The processor addresses the device and transfers an 8-bit data byte from the device to the second operand location.

Condition Code

C	V	G	L	
C	0	0	0	Operation successful
0	1	0	0	Instruction time-out (FALSE SYNC)

Programming Notes

In the RR format, the 8-bit data byte replaces bits 24:31 of the register specified by R2. Bits 0:23 of the register are forced to zero.

These instructions are privileged operations.

Instruction time-out does not prevent the second operand location from being modified.

9.9.4 Read Halfword (RH, RHR)

Read Halfword (RH)
Read Halfword Register (RHR)

Assembler Notation		Op-Code	Format
RH	R1,D2(X2)	D9	RX1,RX2
RH	R1,A2(FX2,SX2)	D9	RX3
RHR	R1,R2	99	RR

Operation

Bits 22:31 of the register specified by R1 contain the 10-bit device address. The processor addresses the device. If the device is halfword-oriented, the processor transfers 16 bits of data from the device to the second operand location. If the device is byte-oriented, the processor transfers two 8-bit bytes in successive operations.

Condition Code

C	V	G	L	
0	0	0	0	Operation successful
0	1	0	0	Instruction time-out (FALSE SYNC)

Programming Notes

If the device is byte-oriented, it must be capable of supplying both bytes without intervening status checks. This instruction does not perform status checking between the two byte transfers.

In the RR format, the data transferred from a halfword device replaces bits 16:31 of the register specified by R2. Bits 0:15 are forced to zero. The first byte of data from a byte device replaces bits 16:23 of the register specified by R2 and the second byte replaces bits 24:31. Bits 0:15 of the register specified by R2 are forced to zero.

In the RX format, the second operand must be located on a halfword boundary. The first byte of data from a byte device replaces bits 0:7 of the halfword operand in memory and the second byte replaces bits 8:15.

These instructions are privileged operations.

Instruction time-out does not prevent the second operand location from being modified.

9.9.5 Write Data (WD, WDR)

Write Data (WD)
Write Data Register (WDR)

Assembler Notation		Op-Code	Format
WD	R1,D2(X2)	DA	RX1,RX2
WD	R1,A2(FX2,SX2)	DA	RX3
WDR	R1,R2	9A	RR

Operation

Bits 22:31 of the register specified by R1 contain the 10-bit device address. The processor addresses the device and transfers an 8-bit data byte from the second operand location to the device. Neither operand is changed.

Condition Code

C	V	G	L	
0	0	0	0	Operation successful
0	1	0	0	Instruction time-out (FALSE SYNC)

Programming Notes

In the RR format, the 8-bit data byte is transferred from bits 24:31 of the register specified by R2.

These instructions are privileged operations.

9.9.6 Write Halfword (WH, WHR)

Write Halfword (WH)
Write Halfword Register (WHR)

Assembler Notation		Op-Code	Format
WH	R1,D2(X2)	D8	RX1,RX2
WH	R1,A2(FX2,SX2)	D8	RX3
WHR	R1,R2	98	RR

Operation

Bits 22:31 of the register specified by R1 contain the 10-bit device address. The processor addresses the device. If the device is halfword-oriented, the processor transfers 16 bits of data from the second operand location to the device. If the device is byte-oriented, the processor transfers two 8-bit data bytes in successive operations.

Condition Code

C	V	G	L	
0	0	0	0	Operation successful
0	1	0	0	Instruction time-out (FALSE SYNC)

Programming Notes

If the device is byte-oriented, it must be capable of accepting both bytes without intervening status checks. This instruction does not perform status checking between the two byte transfers.

In the RR format, data is transferred to a halfword device from bits 16:31 of the register specified by R2. The first byte of data is transferred to a byte device from bits 16:23 of the register specified by R2; the second byte comes from bits 24:31.

In the RX format, the second operand must be located on a halfword boundary. The first byte of data is transferred to a byte device from bits 0:7 of the halfword operand in memory and the second byte is transferred from bits 8:15.

These instructions are privileged operations.

9.9.7 Autoload (AL)

Assembler Notation		Op-Code	Format
AL	D2(X2)	D5	RX1,RX2
AL	R1,D2(X2)	D5	RX1,RX2
AL	A2(FX2,SX2)	D5	RX3
AL	R1,A2(FX2,SX2)	D5	RX3

Operation

The AL instruction loads memory with a block of data from an input device. The 8-bit input device address is specified by memory location X'000078'. The device command byte is specified by memory location X'000079'.

If the R1 field of this instruction is not specified, or contains zero, the default value X'000080' is used for the start address of the data block in memory and the second operand address is used for the end of the data block. If the R1 field of this instruction contains a value other than zero, then the contents of the general registers specified by R1 and R1+1 are used for the start and end of the data block, respectively. If the start address is greater than the end address, the instruction is aborted.

The address of a selector channel is specified by memory location X'00007D'. If the byte at this location contains zero, the selector channel is not used by this instruction. In this case, data is transferred a byte at a time from the input device to successive memory locations, beginning with the specified block start address. If any blank or zero bytes are input before the first nonzero byte, these bytes are considered to be leader and are ignored. All other zero bytes are stored as data. When a data byte has been stored at the specified block end address, the instruction terminates.

If the selector channel address specified by memory location X'00007D' is not zero, the selector channel is used to transfer data from the input device to successive memory locations, beginning with the specified block start address. All data bytes are transferred; no checking for leading zero bytes can be made. The instruction terminates when data has been stored at the specified block end address.

Condition Code

C	V	G	L	
0	0	0	0	Operation successful or aborted
X	1	X	X	Examine status or time out
X	X	1	X	End of medium
X	X	X	1	Device unavailable

Programming Notes

This instruction may be used only with devices whose addresses are less than, or equal to, X'FF'.

This instruction is a privileged operation.

Bad status termination results if any of the least significant three bits of the device status are set.

If the selector channel is not used, the starting and ending addresses for this instruction are relocatable. Address translation should be disabled before attempting to use this instruction in this case.

If the selector channel is used, the data block must begin on a halfword boundary and end on an odd byte boundary. The block may be loaded anywhere in memory. Software must issue a stop command to the selector channel following this instruction. A selector channel interrupt may be queued by this instruction.

If the R1 field of this instruction is used, it must specify the even member of an even/odd register pair.

9.9.8 Simulate Channel Program (SCP)

Assembler Notation		Op-Code	Format
SCP	R1,D2(X2)	E3	RX1,RX2
SCP	R1,A2(FX2,SX2)	E3	RX3

Operation

The second operand address is the address of a Channel Command Block (CCB). The buffer switch bit of the Channel Command Word (CCW) specifies the buffer to be used for the data transfer. If this bit is set, buffer 1 is used. If it is zero, buffer 0 is used. If the byte count field of the current buffer is greater than zero, the V flag in the condition code is set, and the next sequential instruction is executed. If the byte count field is not greater than zero, the following data transfer operation is performed.

If the CCW specifies read, a byte of data is moved from bits 24:31 of the register specified by R1 to the appropriate buffer location. If the CCW specifies write, a byte of data is moved from the appropriate buffer location to bits 24:31 of the register specified by R1. Bits 0:23 are forced to zero.

After a byte has been transferred, the count field of the appropriate buffer is incremented by one. If the count field is now greater than zero, and if the fast bit of the CCW is zero, the buffer switch bit of the CCW is complemented.

Condition Code

C	V	G	L	
0	0	0	0	Count field is now zero
0	0	0	1	Count field is now less than zero
0	0	1	0	Count field is now greater than zero
0	1	0	0	Count field was greater than zero

Programming Notes

If the CCW specifies fast mode, buffer 1 may be used, but the buffer bit is not switched when the count field becomes greater than zero.

The second operand must be located on a fullword boundary.

This instruction is a privileged operation.

9.10 AUTO DRIVER CHANNEL

The auto driver channel provides a means for multiplexing block data transfers between memcry and low or medium speed I/O devices. The channel operaticn is similar, in some respects, to interrupt driven I/O. The channel is activated as a result of a service request from a device on the I/O bus. Upon receipt of such a request, the processor uses the device number to index into the interrupt service pointer table. If the value contained in the table is even, the prccessor transfers control to the interrupt service routine. If the value is odd, it transfers control to the auto driver channel.

To the auto driver channel, the address in the interrupt service pointer table is the address plus one (making it odd) of a Channel Command Block (CCB). The channel command block is a channel program consisting cf a description of the operation to be performed, and a list of parameters associated with the operation. In addition tc the functions of read and write, the channel can also:

1. translate characters
2. test device status
3. chain buffers
4. calculate longitudinal and cyclic redundancy check values
5. transfer control tc software routines to take care of unusual situations

9.11 CHANNEL COMMAND BLOCK

9.11.1 Introduction

The Channel Command Block (CCB), as shown in Figure 9-1, consists of a channel command word (16 bits) that describes the function; count fields (16 bits each) for two buffers; final addresses (32 bits each) for two buffers; a check word (16 bits) for the longitudinal or cyclic redundancy check; the address (32 bits) of a translation table; and the address (16 bits) of a software routine. The CCB requires 22 bytes of memory.

Many interrupt service routines may be available at any time to service device requests. There may also be many channel command blocks in the system ready to handle data transfers as required. Each channel command block must be aligned on a fullword boundary. The channel command block address, plus one, must be placed in the interrupt service pointer table location for the device involved in the transfer.

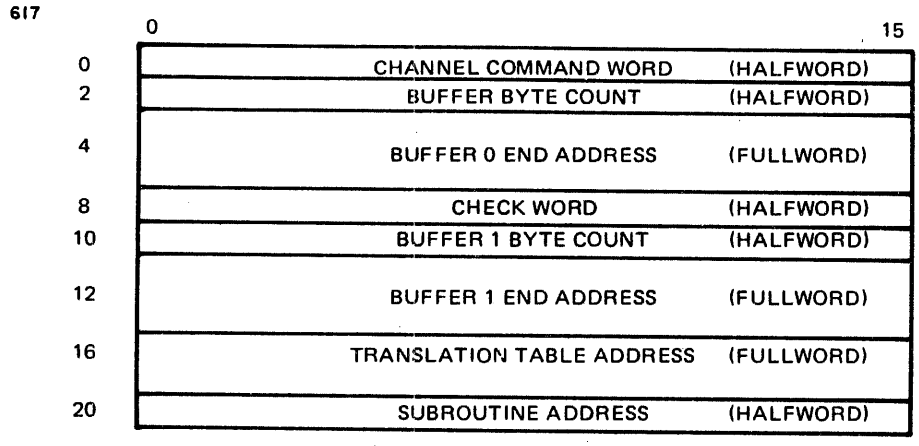


Figure 9-1 Channel Command Block

9.11.2 Subroutine Address

To handle special situations, channel control is transferred to the software subroutine, whose address is contained in the channel command block. When this occurs, registers 0:4 of the appropriate set have already been set up by the processor to contain the old PSW, the device number, the device status, and the address of the channel command block. The current PSW status specifies run state, machine malfunction interrupt enabled, higher level I/O interrupts enabled, and all other interrupts disabled.

The channel transfers control to the subroutine either unconditionally (controlled by a bit in the channel command word), because of bad device status, because of special character translation, or because it has reached the limit of a buffer. It indicates its reason for transferring control by adjusting the condition code as follows:

C	V	G	L	
0	0	0	0	Unconditional transfer or special character
0	0	0	1	Bad status
0	0	1	0	Buffer limit

The subroutine address in the CCB is a 16-bit physical address. For this reason, the subroutine at that address, or at least the first instruction of the subroutine, must reside in the 64 kb of memory.

9.11.3 Buffers

There is a space in the CCB tc describe two data buffer areas. The data areas may be located anywhere in memory. The limits of each data area are described by an address field and a count field. The address field contains the physical address of the last byte in the data area. This address is right justified in the fullword provided. If the device being controlled is a halfword-oriented device, the final address must be odd. If the device is a byte-oriented device, the address may be either odd or even. The active buffer is selected by a bit in the channel command word. When one buffer has been exhausted, the channel may reverse the state of this bit and thus switch to the alternate buffer. Automatic buffer switching is available only for byte-oriented devices and if the Fast bit of the CCW is zero. If the Fast bit is set, buffer 0 is always used.

The count field, in most operations, contains a negative number whose absolute value is equal to one less than the number of bytes to be transferred. The one exception is the case of a single data transfer, for which the count field contains zero.

During data transfers, the channel adds the value contained in the count field to the final address in order to obtain the current address. It makes the transfer, using the current address, then increments the value in the count field by one for a byte device or by two for a halfword device. When the count field becomes greater than zero, the channel sets the G flag in the condition code and transfers control to the specified software subroutine. If the count field is greater than zero upon channel activation, the channel makes no transfer and relinquishes control of the processor.

9.11.4 Translation

The translation feature is available only for byte-oriented devices and if the Fast (F) bit in the CCW is zero. If translation is specified, the fullword provided in the channel command block must contain the address, right justified, of a translation table. This table, which must be aligned to a halfword boundary, can contain up to 256 halfword entries. The format of this table is identical to that used by the Translate (TLATE) instruction (see Section 3.3.2). During data transfers, the channel multiplies the data byte by two and adds this value to the translation table address. The result is the address within the translation table cf the halfword entry corresponding to the data byte.

The channel tests this entry, and, if bit 0 of the halfword is set, it substitutes bits 8:15 of the halfword for the data byte and proceeds with the operation. If bit 0 of the halfword is a zero, the channel:

- does not increment the byte count for the appropriate buffer.
- puts the data byte, untranslated, in bits 24:31 of register 3, of the appropriate set, and forces bits 0:23 of register 3 to zero.
- multiplies the value contained in the translation table by two, and transfers control to the software special character translation routine located at the resulting address.

  Upon transfer to the translation subroutine, registers 0 and 1 contain the old PSW; register 2 contains the device number; register 3 contains the untranslated character; and register 4 contains the address of the channel command block. The current PSW indicates run state, machine malfunction interrupt enabled, higher level I/O interrupts enabled and all other interrupts disabled. The condition code is zero.

9.11.5 Check Word

The check word in the channel command block contains the accumulated residual for longitudinal or cyclic redundancy checking. The initial value for the check word is usually zero. (There are data dependent exceptions, e.g., where initial characters are not to be included in the check.)

The longitudinal check is an exclusive OR of the character with the check word.

The cyclic check uses the formula for CRC 16:

$$X^{16} + X^{15} + X^{2} + 1$$

If the data communication option is equipped, the cyclic check may optionally use the formula for CRC SDLC:

$$X^{16} + X^{12} + X^{5} + 1$$

On input, if both redundancy checking and translation are required, the character is translated first; then the cyclic redundancy check is done using the original character input rather than the translated character. On output, the translated character participates in the redundancy check. Redundancy checking may be used only with byte devices, and is only performed if the Fast bit (F) of the CCW is zero.

9.11.6 Channel Command Word

The Channel Command Word (CCW), as shown in Figure 9-2, consists of two parts. Bits 0:7 contain a status mask. Bits 8:15 describe the channel operation.

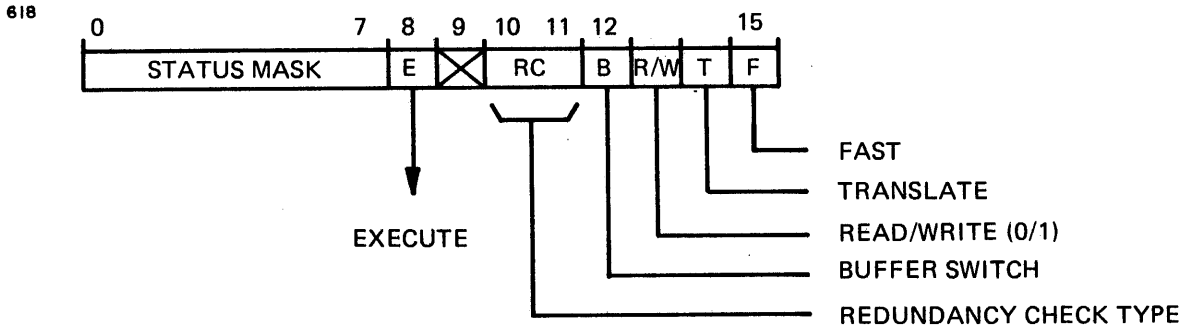


Figure 9-2 Channel Command Word

Status Mask

On every channel operation, if the Execute (E) bit is set, the status mask is ANDed with the device status. This operation does not change the status mask. If the result is zero, the channel proceeds with the operation. If the result is nonzero, the channel sets the L flag in the condition code, and transfers control to the specified software subroutine.

Execute Bit (E)

If this bit is zero, the channel unconditionally transfers control to the specified subroutine, without taking any other action. The condition code is zero. If this bit is set, the channel continues with the operation as specified in the channel command word.

Fast Bit (F)

If this bit is set, the channel performs the I/O transfer in the fast mode. In this mode, buffer switching, redundancy checking, and translation are not allowed. This bit must be set for halfword devices. If this bit is set, buffer 0 is always used.

Read/Write Bit (R/W)

This bit indicates the type of operation. If this bit is zero, a byte or a halfword is input from the device. If this bit is set, a byte or a halfword is output to the device.

Translate Bit (T)

If this bit is set, and the Fast bit is zero, the channel translates the data byte, using the translation table defined in the CCB.

Redundancy Check Type Bits (RC)

These two encoded bits specify the type of redundancy check required. No check is performed if the Fast bit is set. CRC SDLC may be performed only if the data communication option is installed. If the option is not installed, CRC BISYNC (CRC 16) is performed when SDLC is specified. The following table contains the valid types of checks:

Bit 10	Bit 11	Redundancy Check Type
0	0	LRC
0	1	CRC BISYNC
1	0	Reserved - must not be specified
1	1	CRC SDLC - Should only be specified if the data communication option is installed.

Buffer Switch Bit (B)

When zero, this bit specifies that buffer 0 is to be used for the transfer. If it is set, buffer 1 is used. The channel chains buffers, when the count field becomes greater than zero, by complementing the buffer switch bit before transferring control to the specified software routine. Buffer 0 is always used if the Fast bit in the CCW is set.

9.11.7 Valid Channel Command Codes

The following is a list of valid codes for the channel command word. Note that only the first three may be used with halfword devices.

CHANNEL COMMAND WORD 8:15

HEXADECIMAL	BINARY	MEANING
00	00000000	Transfer to subroutine
81	10000001	Read fast mode
85	10000101	Write fast mode
80	10000000	LRC, Buffer 0, read
82	10000010	LRC, Buffer 0, read, translate
84	10000100	LRC, Buffer 0, write
86	10000110	LRC, Buffer 0, write, translate
88	10001000	LRC, Buffer 1, read
8A	10001010	LRC, Buffer 1, read, translate
8C	10001100	LRC, Buffer 1, write
8E	10001110	LRC, Buffer 1, write, translate
90	10010000	CRC BISYNC, Buffer 0, read
92	10010010	CRC BISYNC, Buffer 0, read, translate
94	10010100	CRC BISYNC, Buffer 0, write
96	10010110	CRC BISYNC, Buffer 0, write, translate
98	10011000	CRC BISYNC, Buffer 1, read
9A	10011010	CRC BISYNC, Buffer 1, read, translate
9C	10011100	CRC BISYNC, Buffer 1, write
9E	10011110	CRC BISYNC, Buffer 1, write, translate
B0	10110000	CRC SDLC, Buffer 0, read
B2	10110010	CRC SDLC, Buffer 0, read, translate
B4	10110100	CRC SDLC, Buffer 0, write
B6	10110110	CRC SDLC, Buffer 0, write, translate
B8	10111000	CRC SDLC, Buffer 1, read
BA	10111010	CRC SDLC, Buffer 1, read, translate
BC	10111100	CRC SDLC, Buffer 1, write
BE	10111110	CRC SDLC, Buffer 1, write, translate

9.11.8 General Auto Driver Channel Programming Procedure (See Figure 9-3)

1. Set up interrupt service pointer table to vector to error routines for undefined devices.

2. Set up address of channel command word + 1 (odd) in table at 2 times device number plus X'D0' (start of interrupt service pointer table).

3. Set up complete channel command block.

4. Set up device and enable device interrupt.

5. Enable I/O interrupts in PSW (auto driver channel performs I/O operation).

6. Check for good termination of auto driver channel operation when the subroutine defined in the CCB is entered.

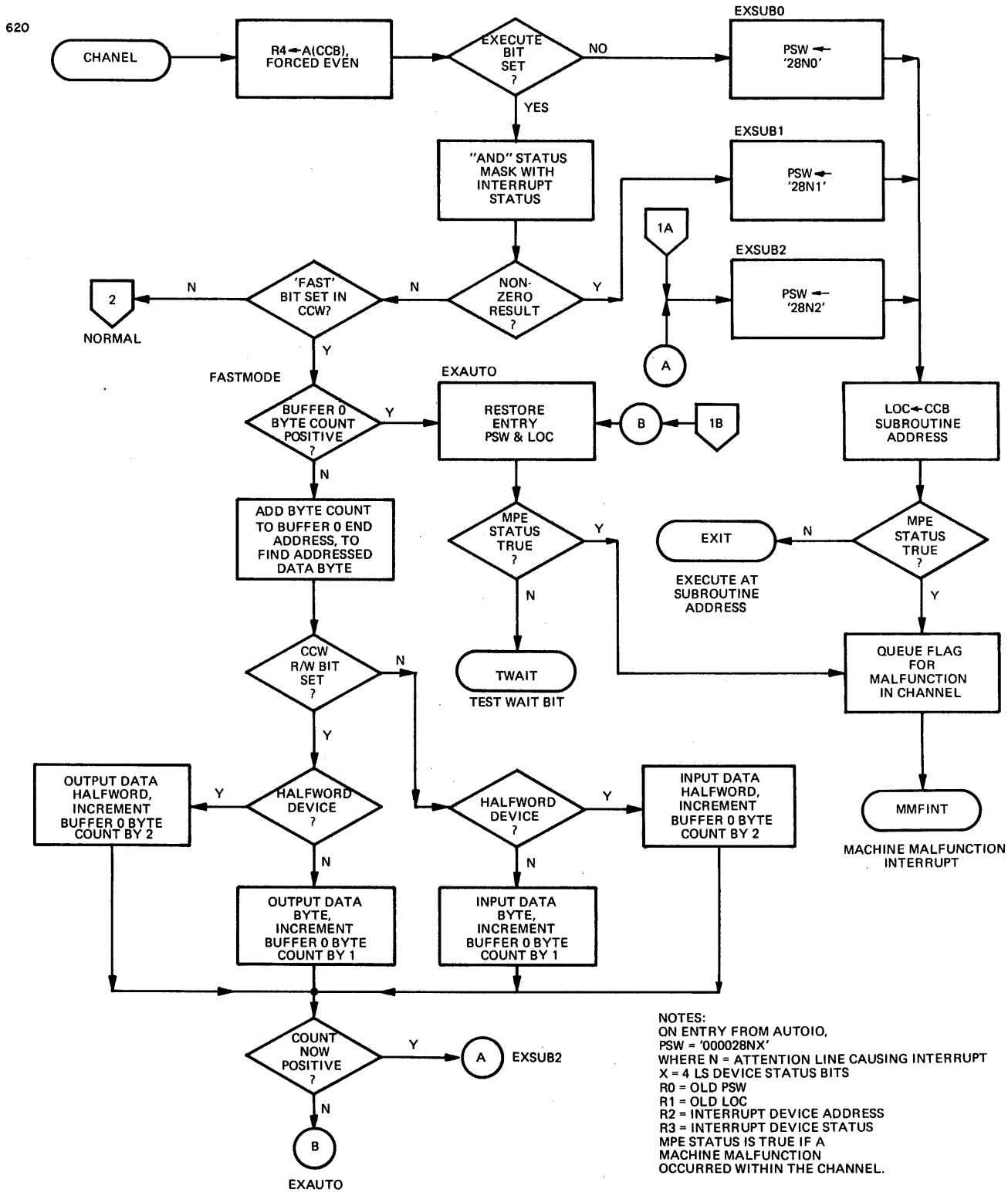


Figure 9-3 Auto Driver Channel Flowchart

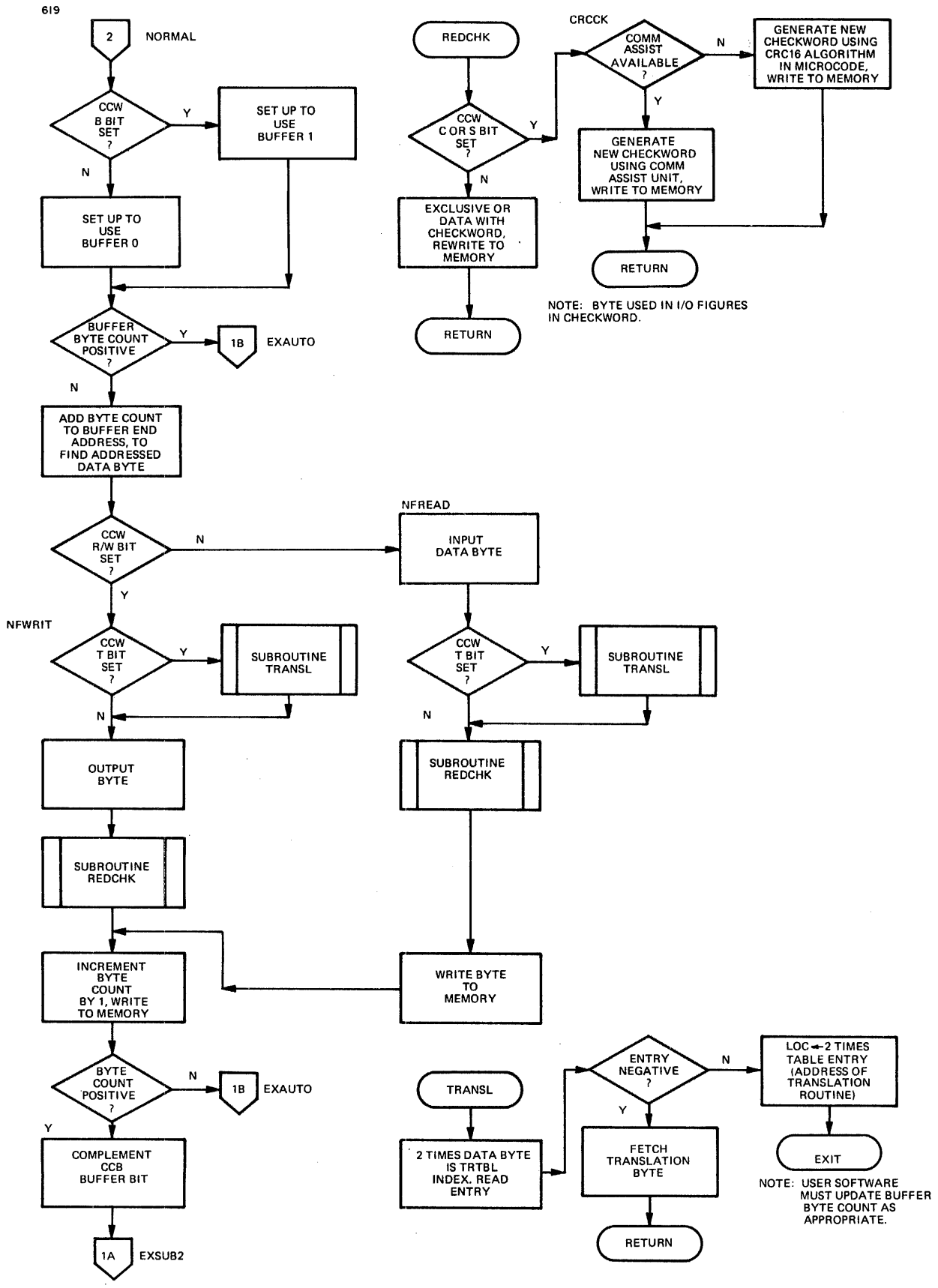


Figure 9-3 Auto Driver Channel Flowchart (Continued)

CHAPTER 10
STATUS SWITCHING AND INTERRUPTS

10.1 INTRCDUCTION

The processor's interrupt system provides a mechanism for escape from the normal processing sequence to handle external and internal events. The software routine that is executed in response to an interrupt is called an interrupt service routine. Before transferring control tc a service routine, the current state of the processor is preserved so that, upon completion of the service routine, the execution of an interrupted program may be resumed.

Interrupts may be classified as being synchronous or asynchronous, depending on whether they occur in fixed relationship to the execution of instructions, or whether they occur at random times due tc events external to the processor. Examples of asynchronous interrupts include power fail, console attention, and peripheral device interrupts.

Synchronous interrupts occur due to fault conditions, or in the case of software interrupts, may be programmed to occur. Examples of fault conditions which cause synchronous interrupts include noncorrectable memory errors, illegal instructions, and arithmetic faults.

Software interrupts occur when the Supervisor Call (SVC) or Simulate Interrupt (SINT) instructions are executed, or as a result of adding an entry tc the system queue. The Breakpoint (BRK) instruction causes program execution to be suspended so that the system console terminal may be activated. See the chapter on the System Console Terminal.

Each interrupt condition is reset when the corresponding interrupt is serviced by the processor.

10.2 PROGRAM STATUS WORD (PSW) AND RESERVED MEMORY LOCATIONS

The Program Status Word (PSW), shown in Figure 10-1, is a 64-bit quantity that controls the operation of the processor. The PSW provides information about various states and conditions affecting the operation of the processor. The PSW is composed of two fullwords: bits 0:31 are the status word, and bits 32:63 are the location counter. The various PSW fields are described below:

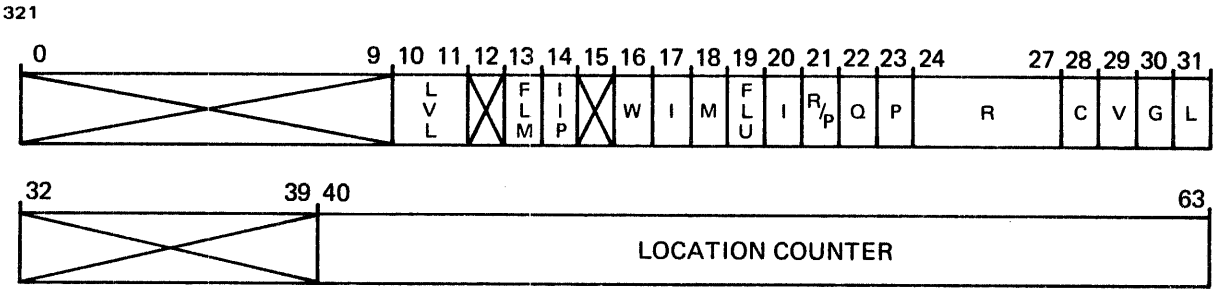


Figure 10-1 Program Status Word (PSW)

Bits 0-9		Unused, must be zero
Bits 10-11	LVL	Memory access level
Bit 12		Unused, must be zero
Bit 13	FLM	Floating-point masked mode
Bit 14	IIP	Interruptible instruction in progress
Bit 15		Unused, must be zero
Bit 16	W	Wait state
Bit 17	I	I/O interrupt mask
Bit 18	M	Machine malfunction interrupt mask
Bit 19	FLU	Floating-point underflow mask
Bit 20	I	I/O interrupt mask
Bit 21	R/P	Relocation/protection mask
Bit 22	Q	System queue service interrupt mask
Bit 23	P	Protect mode
Bits 24 - 27	R	Register set select field
Bits 28 - 31	C,V,G,L	Condition code
Bits 32 - 39		Unused, must be zero
Bits 40 - 63		Location counter

10.2.1 Program Status Word (PSW)

Bits 0:31 of the PSW are called the status word. This word controls interrupts, defines the status of the processor, and contains the condition code. The following sections provide detailed definitions of various states of the processor and how the status word controls them. Unused bits of the status word must always be zero.

10.2.1.1 Memory Access Level Field (LVL)

When PSW bit 21 (R/P) is set, PSW bits 10 and 11 participate in an access level check for any memory access attempted by the current program. The LVL field of PSW is compared numerically to the access level field of the appropriate segment table entry. If the LVL field contains a lesser value than the access level field, a Memory Address Translator (MAT) fault interrrupt occurs.

When PSW bit 21 is zero, PSW bits 10 and 11 are ignored, and no access level check is performed.

10.2.1.2 Floating-Point Masked Mode (FLM)

On processors with the floating-point option, when bit 13 of the current PSW is zero, a program may execute any legal floating-point instruction.

When bit 13 of the current PSW is set, the processor is in the Floating-Point Masked (FLM) mode. A program running in this mode is not allowed to execute floating-point arithmetic instructions. If execution of any floating-point arithmetic instruction is attempted in FLM mode, an illegal instruction interrupt occurs. If the processor is in FLM mode when a context switch is made by the system program and the processor state must be saved, the contents of the floating-point registers need not be saved. This results in a faster context switch.

10.2.1.3 Interruptible Instruction in Progress (IIP)

PSW bit 14 is set by the processor while an interruptible instruction is in progress, and is zero when the interruptible instruction terminates. This bit is set by the processor to indicate that the scratchpad registers contain valid parameters for the interruptible instruction and that these parameters need not be recalculated before resuming the interrupted instruction.

If bit 14 of the current PSW is set when the processor transfers control to a software interrupt service routine, that routine must not allow the contents of the scratchpad registers to be modified before the interruptible instruction is resumed. The STPS, LDPS, ISSV, and ISRST instructions provide the means for saving and restoring these registers if they must be used by the interrupt service routine.

10.2.1.4 Wait State (W)

When PSW bit 16 is set, the processor is in the wait state. In the wait state, the normal fetch instruction/execute instruction/fetch next instruction sequence is suspended. While in the wait state, the processor is responsive to console attention interrupts and primary power fail, as well as any interrupts specifically enabled by the current PSW.

PSW bit 16 is zero when the processor is executing instructions. This bit is forced to zero whenever the single-step, run switch, or system console terminal is used to initiate instruction execution. This bit is not forced set by entry to the console mode.

If an interrupt occurs, PSW bit 16 is set according to the new PSW defined for servicing the interrupt. Bit 16 of the new PSW for any I/O interrupt is zero.

Except for an I/O interrupt, the state of bit 16 of the new PSW is tested as the PSW is loaded. If bit 16 of the newly loaded PSW is set, the processor enters the wait state, provided that no interrupt is still pending. All pending interrupts are serviced before the processor enters the wait state.

10.2.1.5 I/O Interrupt Mask (I)

PSW bits 17 and 20 are used together to enable or disable recognition of interrupt requests generated by peripheral devices on any of the four interrupt levels, as detailed below:

BIT 17	BIT 20	MEANING
0	0	All levels disabled
0	1	Higher levels enabled
1	0	All levels enabled
1	1	Current and higher levels enabled

The interrupt levels are numbered from 0 to 3, with level 0 being the highest priority interrupt level and level 3 being the lowest priority interrupt level.

An I/C interrupt request is queued until the processor acknowledges the interrupt unless the request is programmed reset, or power fail occurs. The state of PSW bits 17 and 20 is ignored by the Simulate Interrupt (SINT) instruction.

10.2.1.6 Machine Malfunction Interrupt Enable (M)

PSW bit 18 is used to enable and disable detection of various malfunction conditions within the processor and the resulting machine malfunction interrupt. When this bit is set, any of the following conditions results in a machine malfunction interrupt.

- early power failure
- power restore
- noncorrectable memory data error
- nonconfigured memory address
- shared memory power failure

The processor is designed with the concept that all software must enable the machine malfunction interrupt for maximum data integrity. Unlike other prccessors, this does not require that this interrupt ever be disabled. The processor resets each detected interrupt condition as it occurs.

While performing a machine malfunction interrupt PSW swap, the processor sets PSW bit 18 tc allow error detection for the new PSW data fetched from memcry. If the new PSW cannot be fetched correctly, the processor effectively stops by entering the console mode. This prevents a runaway situation in the event of a double fault.

If PSW bit 18 is zero, any noncorrectable memory data error is logged by the optional error logger. Cache accesses to memory using a nonconfigured memory address result in undefined data being loaded into the optional high-speed cache, with no error indication. (Subsequent access to the same area of cache results in another memory fetch, as cache data is invalidated.) No machine malfunction interrupt occurs for any of the reasons given above. A machine malfunction due to early power failure or shared memory power failure is queued until PSW bit 18 is set by software, or until automatic shutdown occurs. The interrupt is not queued for any other reascn.

10.2.1.7 Floating-Point Underflow Interrupt Enable (FLU)

PSW bit 19 controls response cf the processor to an arithmetic underflow resulting from a single- or double-precision floating-point arithmetic operation.

If this bit is set when the underflow occurs, an arithmetic fault interrupt occurs, and the participating floating-point registers remain unchanged.

If this bit is zero when the underflow occurs, the result of the operation is replaced by zero, and the condition code is set to 0100 (V-flag only), as defined in the description of the specific floating-point instruction.

10.2.1.8 Relocation/Protection Enable (R/P)

PSW bit 21 is used to enable and disable the relocation and protection programmed into the Memory Address Translator (MAT). When this bit is set, relocation, protection, and the MAT fault interrupt are enabled. When this bit is zero, relocation, protection, and the MAT fault interrupt are disabled.

10.2.1.9 System Queue Service Interrupt Enable (Q)

If bit 22 of the new PSW loaded by any of the instructions listed below is set, the state of the system queue is tested. If the system queue is not empty, a System Queue Service (SQS) interrupt occurs. If the system queue is empty, the next instruction is fetched and executed, according to the newly-loaded PSW.

If bit 22 of the newly-loaded PSW is zero, the SQS interrupt is disabled.

The following instructions test the state of the system queue:

MNEMONIC	MEANING
EPSR	Exchange Program Status Register
LDPS	Load Process State
LPSW	Load Program Status Word
LPSWR	Load Program Status Word Register

10.2.1.10 Protect Mode Enable (P)

When PSW bit 23 is set, the processor is in the protect mode. Any attempt by a program running in this mode to execute a privileged instruction causes an illegal instruction interrupt to occur. The processor does not attempt to execute the offending instruction. The Breakpoint (BRK) instruction is a privileged instruction.

When PSW bit 23 is zero, the processor is in privileged mode. A program running in privileged mode may execute any legal instruction, within the constraints imposed by the system configuration and the state of PSW bit 13 (FLM).

10.2.1.11 Register Set Select Field (R)

Bits 24, 25, 26, and 27 of the current PSW select the active general register set. Although 16 different sets may be specified by using the four bits of this field, only eight sets of general registers are implemented in this processor. The implemented sets are numbered 0, 1, 2, 3, 4, 5, 6, and 15.

Set 0, 1, 2, or 3 is automatically selected by the processor in handling an I/O interrupt on the corresponding interrupt level. Registers 0 through 4 of that set are used to maintain information pertaining to an I/O interrupt request which is acknowledged on the I/O interrupt level corresponding to the selected register set. Therefore, sets 0, 1, 2, and 3 should not be used for general purpose processing. These sets may, however, be used for processing internal interrupts, which use registers 11 through 15 of the selected set to maintain information pertaining to the interrupt.

Sets 4, 5, 6, and 15 may be allocated according to processing needs, without special consideration. Sets 7 through 14 are not implemented. An attempt to select a set which is not implemented may result in the selection of any set, without any special indication of the error.

When a new PSW is loaded, the specified register set becomes the active set for the next instruction executed.

PSW BIT				SELECTED REGISTER SET
24	25	26	27	
0	0	0	0	0
0	0	0	1	1
0	0	1	0	2
0	0	1	1	3
0	1	0	0	4
0	1	0	1	5
0	1	1	0	6
1	1	1	1	15

10.2.1.12 Condition Code (C, V, G, L)

PSW bits 28:31 contain the condition code. As part of the execution of certain instructions, the state of the condition code may be updated to reflect the nature of the result. Not all instructions affect the condition code.

For most interrupts, bits 28:31 of the new PSW are simply copied to the condition code. For immediate interrupts, the least significant four bits of the status byte for the interrupting device are copied to the condition code after the new PSW has been loaded. No restrictions are imposed on the condition code field of a new PSW contained in a memory location or register. Any condition code value may be specified.

The condition code of the current PSW may be tested by the conditional branch instructions described in Chapter 4.

10.2.2 PSW Location Counter (LOC)

PSW bits 32:63 comprise the location counter, which contains the address of the instruction currently being executed by the processor. When the current instruction is successfully completed, the value contained in the location counter is incremented by the length of the instruction in bytes, and the instruction at the resulting address is fetched.

An instruction which results in a branch being taken causes the contents of the location counter to be replaced with the effective branch address; i.e., with the address of the instruction to which control is to be transferred. The instruction at the new address is the next instruction to be fetched and executed.

When an interrupt occurs, the entire PSW, bits 0:63, is replaced. If bit 16 of the new PSW (the wait bit) is set, the instruction indicated by the new contents of the location counter is not fetched. Manual intervention is required to cause the wait bit to be zero, and the instruction to be fetched and executed. If an interrupt causes the PSW with the wait bit set to be replaced by another new PSW that has the wait bit zero, the instruction indicated by the location counter of that new PSW is fetched and executed.

If an instruction has not been successfully completed when an interrupt PSW swap occurs, the 64-bit PSW, in effect for the instruction being executed at the time of the interrupt, is saved before the interrupt handler is entered. The location counter in the saved PSW points to the instruction being executed at the time the interrupt occurred. If the interrupt occurs after the successful completion of one instruction and before beginning another, the location counter in the saved PSW points to the next instruction to be executed.

See Section 10.5, Status Switching, for an explanation of old, current, and new PSW, and of the use of these PSWs by the processor in scheduling interrupt service routines.

10.2.3 Reserved Memory Locations

Physical memory locations X'000000' - X'0002CF' are reserved memory locations. For systems with expanded I/O interrupt service pointer tables, physical memory locations X'0002D0' - X'0004CF' or X'0002D0' - X'0008CF' are also reserved memory locations. These locations contain assorted information used in servicing interrupts, as shown in Figure 10-2. Use of data in these locations as the result of an interrupt is detailed in the section describing the interrupt.

X'000000' - X'00001F'	Reserved, must be zero
X'000020' - X'000027'	Machine malfunction interrupt old PSW
X'000028' - X'00002B'	Used by console service microcode
X'00002C' - X'00002F'	LM effective address word
X'000030' - X'000037'	Illegal instruction interrupt new PSW
X'000038' - X'00003F'	Machine malfunction interrupt new PSW
X'000040' - X'000043'	Machine malfunction status word
X'000044' - X'000047'	Machine malfunction virtual (program) address word
X'000048' - X'00004F'	Arithmetic fault interrupt new PSW
X'000050' - X'00007F'	Bootstrap loader and device definition table
X'000080' - X'000083'	System queue pointer
X'000084' - X'000087'	Power fail save area pointer
X'000088' - X'00008F'	System queue service interrupt new PSW
X'000090' - X'000097'	Relocation/Protection (MAT fault) new PSW
X'000098' - X'00009B'	Supervisor call new PSW status word
X'00009C' - X'0000BB'	Supervisor call new PSW location counter values
X'0000BC' - X'0000C7'	Reserved, must be zero
X'0000C8' - X'0000CF'	Data format fault new PSW
X'0000D0' - X'0002CF'	Interrupt service pointer table
X'0002D0' - X'0004CF'	Expanded interrupt service pointer table
X'0004D0' - X'0008CF'	Expanded interrupt service pointer table

Figure 10-2 Reserved Memory Locations

10.3 INTERRUPT TIMING AND PRIORITY

10.3.1 Maskable and Nonmaskable Interrupts

Maskable interrupt conditions are controlled by bits in the PSW. When a request to interrupt due to a maskable condition occurs, the corresponding control bit in the PSW is examined. If the control bit indicates that the interrupt is enabled, an interrupt is taken and control is transferred to the appropriate service routine. The paragraph describing each interrupt provides details about the control bit(s), how the interrupt is enabled or disabled, and the effects of enabling or disabling an interrupt.

Nonmaskable interrupts are those which have no corresponding control bits in the PSW. Examples of nonmaskable interrupts are SVC, SINT, Illegal Instruction, and Console Attention. Sections describing each interrupt provide further details.

Figure 10-3 shows the various maskable and nonmaskable interrupts.

622-2

NOTES

(a) NUMBERS IN CIRCLES INDICATE THE PRIORITY OF INTERRUPTS. 1 REPRESENTS THE HIGHEST PRIORITY.

(b) FAULTS ABORT THE CURRENT INSTRUCTION. THE OLD PSW POINTS TO THE FAULTING INSTRUCTION. OTHER INTERRUPTS ARE RECOGNIZED AT THE END OF THE CURRENT INSTRUCTION AND OLD PSW POINTS TO THE FOLLOWING INSTRUCTION.

(c) SYNCHRONOUS INTERRUPTS ARE RECOGNIZED AS THEY OCCUR. ASYNCHRONOUS INTERRUPTS ARE RECOGNIZED BETWEEN THE COMPLETION OF CURRENT INSTRUCTION AND THE INITIATION OF THE NEXT INSTRUCTION.

(d) SQS MAY OCCUR ONLY AS PART OF THE LPSW, LPSWR, EPSR, AND LDPS INSTRUCTIONS.

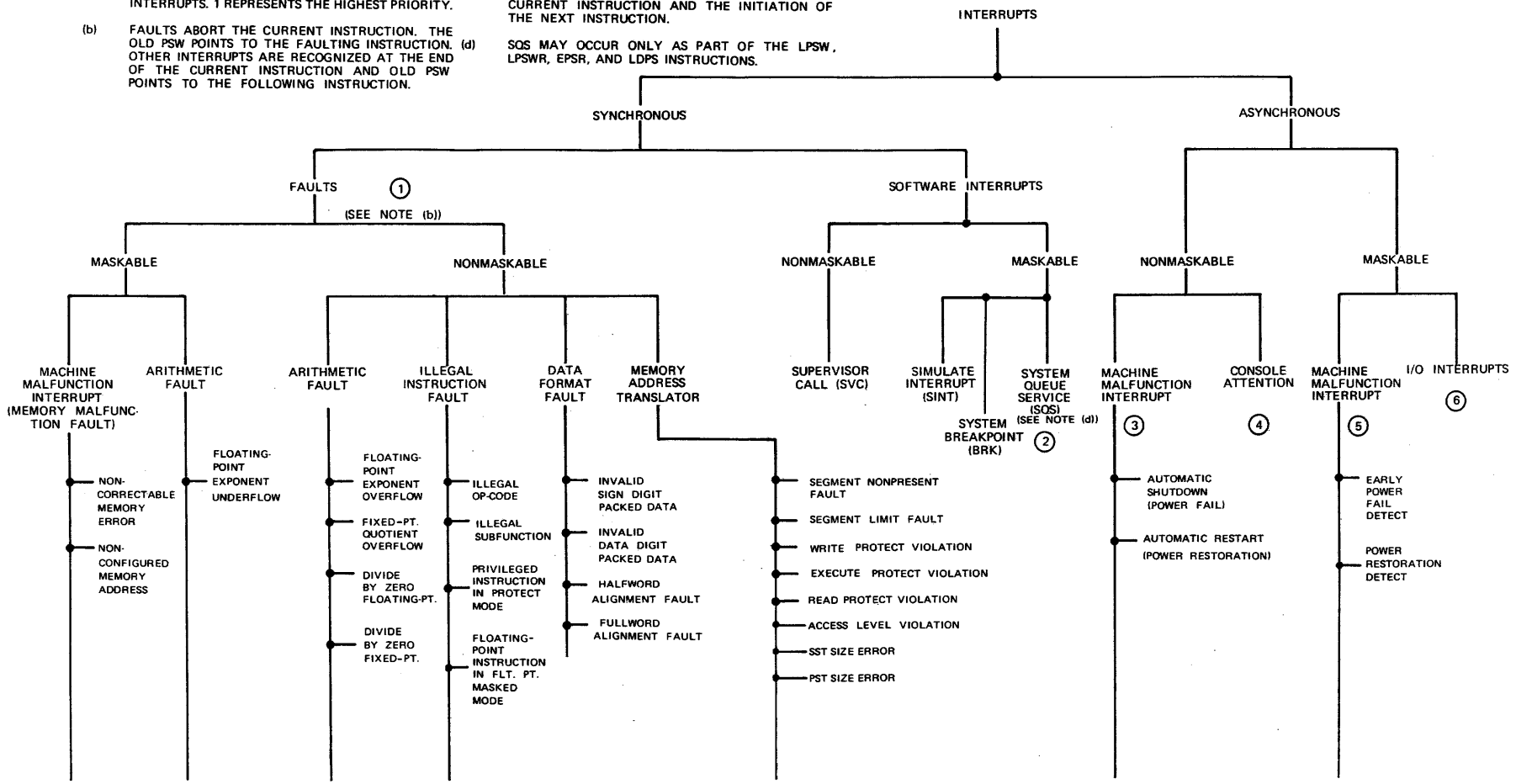


Figure 10-3 Schematic Diagram of Interrupt System Architecture

10.3.2 Interrupt Timing

Asynchronous interrupts are normally permitted to occur only after execution of an instruction has been completed, and before execution of the next instruction begins. However, asynchronous interrupts are permitted to occur at the end of any iteration, while an interruptible instruction is being executed.

A synchronous interrupt is permitted to occur at the time the condition causing the interrupt is detected. The SQS interrupt, which occurs at some indefinite time following addition of an entry to the system queue, is called a deferred synchronous interrupt. A synchronous interrupt due to a fault causes the offending instruction to be aborted with no modification of the contents of registers or memory locations generally resulting from execution of that instruction. Fixed- and floating-point Load/Store Multiple, and Store Double-Precision are exceptions to this rule. In the case of an interruptible instruction, the current iteration of the instruction is aborted by such an interrupt without modification of the contents of registers or memory as a result of the faulted iteration.

For all interrupts, the old PSW location counter presented to the interrupt handler points to the next logically-executed instruction in the interrupted program. If the interrupt is caused by a fault, the instruction causing the fault was not completed and is logically the next instruction to be executed. The old PSW location counter presented to the fault interrupt service routine, therefore, always points to the instruction which caused the fault.

Multiple memory accesses are required for the manipulation of a circular list structure using the ATL, ABL, RTL, or RBL instruction. For each of these instructions, the list header is not updated until the body of the list has been successfully accessed. For the RTL and RBL instructions, no registers are modified unless the list element has been successfully accessed, and the list header has been successfully updated.

10.3.3 Interrupt Precedence

Considering the instant of instruction fetch request as the time of reference, interrupts have the following precedence (highest to lowest):

INTERRUPT PRECEDENCE TABLE

Synchronous Interrupts	Fault interrupts
	System queue service
Asynchronous Interrupts	Primary power fail/restore
	Console attention
	Early power fail
	Shared memory power fail
	I/O interrupts

Fault interrupts are caused by various conditions that have the following logical precedence in descending priority order.

- Relocation/protection fault on an instruction fetch
- Machine malfunction fault due to memory malfunction on an instruction fetch
- Illegal instruction fault
- Illegal subfunction fault
- Data format fault due to alignment error on a data read/write operation
- Relocation/protection fault on a data read/write operation
- Machine malfunction fault due to memory malfunction on a data read/write operation
- Data format fault for other than boundary alignment error
- Arithmetic fault

For a memory malfunction, a nonconfigured memory address fault takes precedence over a noncorrectable memory data fault.

Since any fault interrupt causes execution of an instruction to be aborted at the point of the fault interrupt condition, no more than one fault interrupt condition can occur at a time. However, other interrupts in the synchronous and asynchronous interrupt classes given in the preceding Interrupt Precedence Table can occur simultaneously. In such a case, the order given in the table above governs the servicing sequence for the interrupts.

10.3.4 Interruptible Instructions

For any interruptible instruction, execution consists of the following phases: instruction fetch, instruction decode, an iterative loop, and termination. An interrupt during any phase of an interruptible instruction does not affect the operation of the instruction. It may simply be reexecuted once the interrupt has been serviced. An interrupt during the iterative phase of the instruction causes the processor to resume the iterative phase when the instruction is reexecuted, as though the interrupt never occurred. If the interrupt was caused by a fault, the iteration which resulted in the interrupt is repeated when the instruction is reexecuted.

To abort an interruptible instruction when it is interrupted, PSW bit 14 must be forced to zero before any subsequent interruptible instruction (except RDCS or WDCS) is attempted.

CAUTION

SOFTWARE MUST NEVER SET PSW BIT 14 UNLESS RESUMING EXECUTION OF THE INTERRUPTIBLE INSTRUCTION THAT CAUSED BIT 14 OF THE PSW TO BE SET. RESUMPTION OF ANY INTERRUPTIBLE INSTRUCTION MUST NEVER BE ATTEMPTED IF THE CONTENTS OF THE SCRATCHPAD REGISTERS ARE NOT KNOWN TO HAVE BEEN PRESERVED BETWEEN INSTRUCTION INTERRUPTION AND RESUMPTION.

10.4 PROCESSOR MODES

At any given time, the processor may be in the console mode or run mode. The single-step mode provides a means for alternating between the console and run modes. Wait and run states only have meaning for the run mode.

10.4.1 Console Mode

While the processor is dedicated to communicating with the system console terminal, it is said to be in the console mode. In this mode, program execution is suspended so that the user may examine and modify the data contained in certain registers and memory locations.

Appendix F provides a flowchart for the console service routine. The console mode may be entered in any of the following ways:

1. The Breakpoint (BRK) instruction is executed by a running program when PSW bit 23 is zero.

2. Execution of an instruction is completed while in the single-step mode.

3. The HALT/RUN switch or the SINGLE switch is depressed momentarily while the processor is in the run mode.

4. Following a system initialization sequence, backup power to memory is found not to have been maintained within regulation, and the ISU is not enabled when the sequence is complete.

5. Following a system initialization sequence, if backup power to memory was maintained within regulation, but the LSU is not enabled and the contents of physical memory location X'000028' indicate that the processor was in the console mode when system initialization occurred.

6. An attempt to fetch a machine malfunction interrupt new PSW results in a non-correctable memory error. In this case, the error code for the initial malfunction is stored in the machine malfunction status word at X'000040', and LOC is loaded with the address of the status word before the console mode is entered.

7. If control has been passed to uninitialized Writable Control Store or an errant WCS microprogram, control can be regained at the system console by enabling the single-step mode via the SINGLE switch, and depressing the HALT/RUN switch.

Note that system initialization occurs when the power supply detects that AC line voltage is failing; when the Initialize (INIT) switch on the consolette is momentarily depressed; or when the key-operated LOCK/ON/STANDBY switch is moved to the STANDBY position. The initialization sequence completes when power is restored to the processor. System initialization resets all pending interrupts for the system console and other I/O devices in the system. DMA operations are also terminated.

While the processor is in the console mode, interrupt conditions are not handled in the same manner as they are if detected during execution of a program.

Interrupt requests for the system console terminal and all other I/O devices remain queued until the run mode is entered. DMA operations are not affected by changing processor modes.

PSW bit 16 is always forced to zero before the run mode is entered from the console mode.

Fault conditions caused by memory accesses while in the console mode are reset when they occur, and do not cause interrupts when the run mode is entered. If a fault condition occurs while attempting to modify a memory location, that location may not be changed. If a fault occurs while attempting to examine a memory location, the console service routine is aborted and restarted.

System initialization, while in the console mode, results in automatic shutdown, with no machine malfunction interrupt due to power failure.

10.4.2 Run Mode

When the processor is not dedicated to communicating with the system console terminal, it is in the run mode. In this mode, program execution is controlled by the contents of the 64-bit Program Status Word (PSW). While the processor is in the run mode, it may be in either the wait state (PSW bit 16 is set), or the run state (PSW bit 16 is zero). In the run state, the processor performs a repetitive fetch instruction/execute instruction/fetch next instruction sequence. In the wait state, this sequence is suspended.

The run mode may be entered in any of the following ways:

1. The 'less than' prompt character (<) is entered from the system console terminal when the processor is in the console mode.

2. The HALT/RUN switch is depressed momentarily while the processor is in the console mode.

3. The LSU is installed and enabled when a system initialization sequence is completed. In this case, the program loaded from the LSU is given control of the processor.

4. The 'greater than' single-step character (>) is entered from the system console terminal when the processor is in the console mode. This causes the instruction to be executed in single-step mode, regardless of the position of the SINGLE switch.

Interrupt conditions cannot cause the processor to enter the run mode from the console mode, with the following two exceptions:

1. An initialization sequence performed while the processor is in the console mode causes a program to be loaded from the enabled LSU. Control of the processor is given to that program.

2. The HALT/RUN switch is depressed momentarily while the processor is in the console mode.

10.4.3 Single-Step Mode

When the SINGLE switch is in the SINGLE position, the processor is in the single-step mode. In this mode, whenever execution of an instruction is completed, the processor leaves the run mode and enters the console mode. Manual intervention is normally required to execute the next instruction.

Interrupts are handled according to the methods detailed in the previous paragraphs. If the processor is in the single-step mode and the run state when an interrupt request occurs, the processor completes the current instruction (or iteration) and then performs the interrupt PSW swap. The first instruction of the interrupt service routine is not executed.

If system initialization occurs while in the single-step mode, any instruction in progress (or the current iteration of an interruptible instruction) completes. When the initialization sequence is complete, a maximum of one instruction is executed before the processor again enters the console mode.

If the processor is in the run state when the SINGLE switch is placed in the SINGLE position, the console mode is entered.

Note that in the single-step mode, PSW bit 16 is always forced to zero before entering the run mode to fetch a user instruction.

NOTE

If interrupts are enabled at the system control terminal interface by software, entering the console mode causes interrupts tc be queued from device X'011' (the write side). Depression of any key at the ccnsole may cause an interrupt tc be queued from device X'010' (the read side).

10.5 STATUS SWITCHING

The PSW that is loaded in the processor, at any given time, is called the current PSW. The register set selected by this PSW, the data contained in the general, floating-point, or scratchpad registers accessible by the user program, and the machine status defined by the PSW collectively constitute the "process state". If the status word or both the location counter and status word are changed, a status switch has occurred. A status switch can be caused explicitly by executing a status switching instruction or may be forced to occur by an interrupt. When the value of the PSW that was current at the time of a status switch is saved, that value is called the old PSW.

The scheduling of interrupt service routines is based upon the concepts of old PSW, current PSW, and new PSW. When an interrupt occurs, the following status switch takes place: the current PSW becomes the old PSW; the new PSW defined for the interrupt is loaded, and becomes the current PSW.

For a status switch resulting from an interrupt, the old PSW is stored in dedicated registers of the set specified by the new PSW defined for the interrupt. The machine malfunction interrupt is the exception to this rule; for this interrupt, the old PSW is stored in dedicated memory locations.

For meaningful processor respcnse to multiple interrupts, it is important that the new PSW defined for a particular interrupt class does not enable interrupts of the same class.

The various interrupts which may occur, and the response of the processor to each interrupt, are described in the following sections.

10.5.1 Illegal Instruction Interrupt

The illegal instruction interrupt occurs if an attempt is made to execute an instruction whose operation code is not one of those permitted by the system. This interrupt may occur for any of the following reasons:

1. The operation code is undefined for the system, or optional equipment necessary to execute the instruction is not present in the system.

2. The operation code has several possible subfunction specifications, and the subfunction specified is undefined.

3. The instruction is a privileged instruction, and PSW bit 23 is set.

4. The instruction is a floating-point instruction, and PSW bit 13 is set.

The illegal instruction interrupt cannot be disabled. The floating-point instructions, high speed data handling instructions, and writable control store instructions require optional equipment, and are therefore optionally illegal. No attempt is made by the processor to execute an illegal instruction.

When an illegal instruction interrupt occurs, the following actions are taken:

1. The current PSW is stored in registers 14 and 15 of the set selected by the illegal instruction interrupt new PSW found in memory at physical address X'000030'.

2. The illegal instruction interrupt new PSW becomes the current PSW.

The old PSW location counter presented to the interrupt service routine in register 15 points to the illegal instruction.

10.5.2 Data Format Fault Interrupt

The data format fault interrupt occurs if the required halfword or fullword alignments are violated for memory accesses, or if it is otherwise determined that data is not properly aligned to the specified fields. Halfword alignment violations are not detected by the processor on memory read. The data format fault interrupt cannot be disabled.

When a data format fault interrupt occurs, the following actions are taken:

1. The current PSW is stored in registers 14 and 15 of the set selected by the data format fault new PSW found in memory at physical address X'0000C8'.

2. Register 13 of the selected set is loaded with a code to indicate the reason for the interrupt, as shown in the following list:

CODE	REASON FOR INTERRUPT
0	Reserved code
1	Reserved code
2	Invalid sign digit, packed data
3	Invalid data digit, packed data
4	Reserved code
5	Reserved code
6	Fullword or Halfword alignment fault

3. If the interrupt was caused by a halfword or fullword alignment fault, register 12 of the selected set is loaded with the nonaligned virtual address causing the fault.

4. The data format fault interrupt new PSW becomes the current PSW.

The old PSW location counter presented to the interrupt service routine in register 15 points to the instruction being executed when the fault occurred. A data format fault causes the current instruction, or the current iteration of an interruptible instruction, to be aborted immediately.

10.5.2.1 Alignment Faults

An attempt to fetch a fullword of data from memory, or to write a fullword of data to memory, using a program address which does not have zeros as its two least significant bits, causes a fullword alignment fault.

An attempt to write a halfword of data to memory, using a program address which does not have zero as its least significant bit, causes a halfword alignment fault. The processor does not distinguish between fullword and halfword alignment faults. An alignment fault cannot occur during an instruction fetch on this processor.

If an alignment fault occurs while attempting to write to memory, the fullword or halfword at the next lower aligned address may be modified.

10.5.2.2 Invalid Digit Faults

If an invalid sign or data digit is encountered while processing numeric string data, it is presumed that the data is not aligned to the specified fields. Additional information may be found in the description of the instruction used to process the numeric string.

10.5.3 Relocation/Protection (MAT) Fault Interrupt

This fault interrupt occurs if an executing program violates any of the relocation and protection conditions programmed into the Memory Address Translator (MAT). MAT error checking and the MAT fault interrupt are enabled when PSW bit 21 is set. MAT faults are not queued.

When a MAT fault interrupt occurs, the following actions are taken:

1. The current PSW is stored in registers 14 and 15 of the set selected by the MAT fault interrupt new PSW found in memory at physical address X'000090'.

2. Register 13 of the selected set is loaded with a code to indicate the reason for the interrupt. This code is copied from the MAT status register while simultaneously resetting the fault.

CODE	REASON FOR INTERRUPT
0	Reserved code
1	Execute protect violation
2	Write protect violation
3	Read protect violation
4	Access level fault
5	Segment limit fault
6	Nonpresent segment
7	SST size exceeded
8	PST size exceeded

3. Register 12 of the selected set is loaded with the program address which caused the fault.

4. If the fault occurred on a data fetch while attempting to load the general registers using the Load Multiple (LM) instruction, register 11 of the selected set is loaded with the effective second operand address calculated at the start of the LM instruction.

5. The MAT fault interrupt new PSW becomes the current PSW.

The old PSW location counter presented to the interrupt service routine in register 15 points to the instruction being executed when the fault occurred. Further information on memory management may be found in Chapter 12.

10.5.4 Machine Malfunction Interrupt

The machine malfunction interrupt occurs when any of the following conditions are detected:

- Early power fail
- Power restore
- Noncorrectable memory data error
- Nonconfigured memory address
- Shared memory power fail

Detection of the listed conditions and the machine malfunction interrupt are enabled when PSW bit 18 is set. Early power fail and shared memory power fail detects are queued until primary power fail occurs if PSW bit 18 is zero. All other malfunction conditions are ignored, and the interrupts are lost.

When a machine malfunction interrupt occurs, the following actions are taken:

1. The current PSW is stored in memory beginning at physical address X'000020'.

2. The Machine Malfunction Status Word (MMSW) at physical address X'000040' is loaded with a code to indicate the reason for the interrupt. Only one bit is set in this code:

BIT NUMBER	REASON FOR INTERRUPT
0	PF - Power failure
1	PR - Power restoration
2	NCD - Noncorrectable memory error during data fetch
3	NCI - Noncorrectable memory error during instruction fetch
4	NCA - Noncorrectable memory error during auto driver channel operation
5	NVD - Nonconfigured memory address during data fetch
6	NVI - Nonconfigured memory address during instruction fetch
7	NVA - Nonconfigured memory address during auto driver channel operation
30	SMP - Shared memory power failure

3. If the interrupt was caused by a noncorrectable memory error, or nonconfigured memory address, the virtual address used for the memory access is stored in the machine malfunction virtual address word at physical address X'000044'. Otherwise, the contents of this word are undefined.

4. If the interrupt was caused by a noncorrectable memory error, or nonconfigured memory address, and the fault occurred on a data fetch while attempting to load the general registers using the LM instruction, the effective second operand address calculated at the start of that instruction is stored in the LM effective address word at physical address X'00002C'. Otherwise, the contents of this word are undefined.

5. The machine malfunction interrupt new PSW found at physical address X'000038' becomes the new PSW.

If the interrupt was caused by executing an instruction, the old PSW location counter presented to the interrupt service routine points to the offending instruction. Otherwise, the old PSW location counter presented to the interrupt service routine points to the instruction to be executed once the interrupt has been serviced.

If the interrupt was caused by executing the LM instruction, bits 2 and 5 of the Machine Malfunction Status Word (MMSW), may be used to determine if any registers were modified before the interrupt occurred. If the old PSW location counter points to an LM instruction, and if bits 2 and 5 of the MMSW are both zero, no registers were modified. If bit 2 or bit 5 of the MMSW is set, then:

1. If the data stored at physical addresses X'000044' and X'00002C' are equal to one another, no registers were modified by the instruction before the fault occurred.

2. If the data stored at physical addresses X'000044' and X'00002C' are not equal to one another, at least one register was modified by the instruction before the fault occurred. The number of registers modified may be determined by taking the difference of the data stored at physical addresses X'000044' and X'00002C', and dividing the result by four.

1322

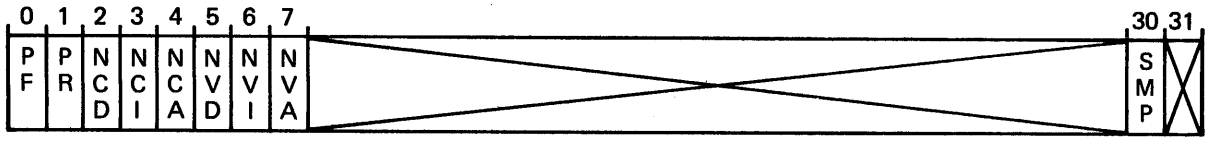


Figure 10-4 Machine Malfunction Status Word (MMSW)

10.5.4.1 Early Power Fail Detect and Automatic Shutdown

Early power fail detect occurs when the primary power failure sensor detects a low voltage; when the power switch is turned from the ON to STANDBY position; or when the INIT switch is depressed.

At the end of execution of the current instruction or the current iteration of the current interruptible instruction, a machine malfunction interrupt is taken if PSW bit 18 is set.

Following early power fail detect, software has one millisecond before the automatic shutdown procedure of the processor takes control as a result of Primary Power Fail. During this procedure, the following actions occur:

1. The fullword power fail save area pointer is fetched from location X'000084'.

2. The following information is saved by firmware in the power fail save area:

DATA	OFFSET IN SAVE AREA (IN BYTES)
Current PSW	0-7
The eight general register sets (in order, 0 through F)	8-519
Interruptible instruction state (scratchpad registers)	520-583
Optional floating-point registers, single and double	584-679

3. The processor waits for power restore.

NOTES

1. If the processor is not equipped with the optional floating-point registers, the area between offsets 584 and 679 is not used.

2. If the pointer found in location X'000084' does not specify a save area aligned to a fullword boundary, the processor forces correct alignment by replacing the two least significant bits of the pointer with zeros.

3. The floating-point masked mode bit in the PSW has no effect on the saving of the floating-point registers.

4. The IIP bit in the PSW has no effect on the saving of the scratchpad registers.

10.5.4.2 Power Restore

When power restore occurs, a simple go/no-go self-test of various internal buses and registers is performed. If the back-up supply voltages to memory were not maintained within margins between shutdown and power restore, the first 256 kbytes of memory are filled with a data pattern tc prevent spurious noncorrectable memory error indications, and the general registers, scratchpad registers, and floating-point registers are loaded with predetermined data.

The first 256 kbytes of memory are then tested to see if data can be held. This test does not modify the data contained in memory. Failure of self-test or the memory test causes that test to loop, as long as the failure persists. During the test, the processor is responsive only to a primary power fail which results in an automatic shutdown, and the FAULT lamp on the consolette switch panel is on.

When memory testing is complete, the FAULT lamp is turned off, and the state of the optional Loader Storage Unit (LSU) is tested. If the LSU is not equipped, it is presumed to be disabled. In all cases, bit 1 of the machine malfunction status word at physical address X'000040' is set to indicate power restore.

10.5.4.2.1 If the LSU is Disabled

If the back-up voltages to memory were not maintained within margins between shutdown and power restore, then memory is assumed not to contain valid data. In this case, a PSW status of '00008000' (wait bit only) and location counter of '00FFFFFE' are loaded and displayed on the system console terminal. Manual intervention is required tc restart the processor. The Memory Voltage Failure (MVF) indication is reset in this case. MVF is discussed under the section "If the LSU is Enabled."

If the back-up voltages to memory were maintained, the data saved in the power fail save area by the automatic shutdown procedure is reloaded.

If the data in memory at physical address X'000028' indicates that the processor was in console mode when power failed, the reloaded PSW is displayed, and communication with the system console terminal resumes.

If the processor was not in console mode when power failed, bit 18 of the reloaded PSW is tested. If the bit is set, a machine malfunction interrupt occurs.

If bit 18 of the reloaded PSW is zero, program execution is resumed using the reloaded PSW. Note that the state of the wait bit (bit 16) of the PSW is tested before executing any instruction.

NOTE

Data in the Memory Address Translator and Selector Channel control registers and Writable Control Store is volatile, and must be considered invalid following any power fail/restore sequence.

10.5.4.2.2 If the LSU is Enabled

After the FAULT lamp is turned off, the program in the LSU is loaded, and control is transferred to it, using the PSW specified in the program. If the memory start address is greater than the memory end address specified for the LSU program, the program is not loaded, and the console mode is entered.

A Memory Voltage Failure (MVF) indication is available to the processor if memory voltages are not maintained within margins between shutdown and power restore. MVF is reset when the console mode is entered, or is reset when the Reset Memory Voltage Failure (RMVF) instruction is executed.

If MVF is indicated following power restore, it is assumed that memory does not contain an executable program. The MVF indication is retained until reset as described above, even if multiple shutdown/power restore sequences occur. Software loaded via the optional LSU should execute the RMVF instruction once the load is complete and all interrupt new PSWs have been established. Proper use of the RMVF instruction prevents a potential runaway condition in the event of multiple power failures.

10.5.4.3 Noncorrectable Memory Error

During write operations to memory, an Error Correcting Code (ECC) is generated. This code enables the memory system to correct any single bit error detected on a subsequent read operation in each fullword of memory. If the operation is only a byte or halfword write to memory, the memory system reads and updates the error correcting code for the fullword of memory that contains the byte or halfword that is being written.

Each time data is read from memory, the error correcting code is re-created and compared to the code generated when data was last written to any part of the fullword memory location. If a data error is detected, and the error is a single bit error, it is corrected transparent to the processor. If, however, a multiple bit error is detected, a memory malfunction fault is generated, since multiple bit errors cannot be corrected.

Note that data with three or more bits in error may not result in a fault. Detection of any error causes a bit to be set in the optional error logger for subsequent readout using the REL instruction.

If PSW bit 18 is zero when the error occurs, the error is ignored, but is logged in the optional error logger.

If PSW bit 18 is set, occurrence of a noncorrectable memory error causes the current instruction (or the current iteration of an interruptible instruction) to be aborted immediately, and a machine malfunction interrupt occurs. Bit 2, 3, or 4 of the machine malfunction status word at physical address X'000040' is set to indicate the reason for the interrupt. The program address used for the memory access is stored in the machine malfunction address word at physical address X'000044'.

If the error occurs on a data fetch while attempting to load the general registers using the LM instruction, the effective second operand address calculated at the start of the LM instruction is stored in the LM effective address word at physical address X'00002C'. This data allows the instruction to be simulated in the event specified index registers were modified.

If the error occurs while fetching an instruction, the old PSW location counter, presented to the interrupt service routine, points to the first halfword of the instruction being fetched.

If the error occurs during an auto driver channel operation, registers 0 and 1 of the set indicated by the old PSW presented to the interrupt service routine contain the PSW for the instruction interrupted by the I/O interrupt that activated the channel. Register 4 of the set indicated contains the address of the CCB that was being executed when the error occurred.

Since the error correcting code is maintained on a fullword basis, if a multiple bit error is detected when a halfword or byte of a fullword is read, it is not possible to determine which bits are in error. Therefore, a reference to any portion of a fullword that contains multiple bit errors may cause a memory malfunction, even though the incorrect bits might not be in the portion of the fullword being accessed. (References to memory made by look-ahead buffers or caches do not cause memory malfunction interrupts until the fullword that is in error is actually used by the currently executing instruction.)

10.5.4.4 Nonconfigured Memcry Address

The processor tests the physical address used for each memory access, if PSW bit 18 is set. When access to memory assigned to a memory controller physically not in the system is attempted, a machine malfunction interrupt occurs. The current instruction (or the current iteration of an interruptible instruction) is immediately aborted. Bit 5, 6, or 7 of the machine malfunction status word at physical address X'000040' is set to indicate the reason for the interrupt. The program address used for the memory access is stored in the machine malfunction address word at physical address X'000044'.

If the error occurs on a data fetch while attempting to load the general registers using the LM instruction, the effective second operand address calculated at the start of the LM instruction is stored in the LM effective address word at physical address X'00002C'. This data allows the instruction to be simulated in the event specified index registers were modified.

If the error occurs while fetching an instruction, the old PSW location counter, presented to the interrupt service routine, points to the first halfword cf the instruction being fetched.

If the error occurs during an auto driver channel operation, registers 0 and 1 of the set indicated by the old PSW presented to the interrupt service routine contain the PSW for the instruction interrupted by the I/O interrupt that activated the channel. Register 4 of the indicated set contains the address of the CCB that was being executed when the error occurred.

10.5.4.5 Shared Memory Power Fail Detect (Optional)

This interrupt can occur only if the processor is equipped with the MOS shared memory option. Shared memory power fail detect occurs when the early power failure sensor in the shared memory power supply detects a low voltage. Each processor attached to the shared memory system is interrupted.

At the end of execution of the current instruction or the current iteration of the current interruptible instruction, a machine malfunction interrupt is taken if PSW bit 18 is set.

Following shared memory power fail detect, software has one millisecond before the MOS shared memory system enters a power-down or stand-by mode. During this time, any necessary housekeeping functions may be performed by system software.

There is no mechanism to indicate to the processor that shared memory power has been restored; however, the following sequence may be performed to test for shared memory power:

1. Store a fullword data pattern in a fullword location in shared memory. This pattern may be any data other than all zeros or all ones.

2. Store a different data pattern in the next-higher-addressed fullword location in shared memory.

3. Compare the contents of the first fullword location with the data pattern stored by step 1. If no error is indicated and the patterns are equal, power is applied to the shared memory system.

Note that if the MOS shared memory back-up supply voltages are maintained within margins between power fail and power restore, data in the shared memory is retained during the power fail/restore sequence. This can be determined by software means. There is no mechanism, however, to indicate this fact to the processor. If the stand-by voltages were not maintained within margins, the data is lost. In this case, software in the master processor must store a fullword data pattern in each fullword of shared memory, initializing the ECC bits to prevent subsequent spurious noncorrectable memory error indications.

An attempt to fetch an instruction or data from shared memory, while the memory is in a power-down or stand-by state, results in a noncorrectable memory error or nonconfigured memory address fault.

10.5.5 Input/Output Device (I/O) Interrupts

10.5.5.1 Priority Levels

Interrupt requests from I/O devices may occur on any of four priority levels. Level 0 is the highest priority level; level 3 is the lowest priority level. Acknowledgement of interrupt requests on the various pricrity levels is enabled by PSW bits 17 and 20, as shown in the following table:

PSW BIT 17	PSW BIT 20	MEANING
0	0	All levels disabled
0	1	Higher priority levels enabled
1	0	All priority levels enabled
1	1	Current and higher priority levels enabled

A unique register set is selected for I/O interrupt requests acknowledged on each pricrity level. For example, when an interrupt request is acknowledged at priority level 3, register set 3 is selected by the processor for handling the interrupt request. If the request results in entry to a software interrupt service routine, register set 3 is selected by the PSW in effect at the time the routine is entered, and information pertaining to the interrupt is contained in registers 0 to 3 or 0 to 4 of that set.

The current priority level is determined by bits 24:27 (the register select field) of the current PSW. For example, if set 3 is currently selected, levels 2, 1, and 0 are higher priority levels, and level 3 is the current priority level. If PSW bit 17 is zero and PSW bit 20 is set, an I/O interrupt request occurring on level 2, 1, or 0 is acknowledged, but a request occurring on level 3 is not acknowledged.

In this example, if PSW bits 17 and 20 are both set (the PSW status is X'4830'), the interrupt request on level 3 is also acknowledged.

If a register set other than 0, 1, 2, or 3 is selected by the current PSW, all I/O interrupt requests are considered to be higher-priority requests, and will be acknowledged if either PSW bit 17 or bit 20 is set.

If the current PSW selects register set 4, 5, 6, or 15, all interrupt levels are considered to be higher priority levels.

Enabling of interrupts on the various levels is shown in detail in Table 10-1. When an interrupt request occurs, but is not acknowledged by the processor, the request remains queued until one of the following occurs:

1. The interrupt request is acknowledged by the processor when enabled by the current PSW.
   when enabled by the current PSW.

2. The interrupt request is programmed reset by the software.

3. System initialization occurs.

When the processor acknowledges an I/O interrupt request, the result may be either an auto driver channel operation, or an immediate interrupt. In either case, the register set associated with the priority level, on which the interrupt is acknowledged, is used in processing the interrupt.

For further information on programming a device interrupt request reset, refer to the programming manual for the specific device. This feature is not available for all I/O devices.

10.5.5.2 Immediate Interrupt - Auto Driver Channel Operation

An interrupt request by an I/O device at one of the four interrupt priority levels is acknowledged only when interrupts are enabled for that level, as defined by the status of PSW bits 17 and 20, and the selected register set.

TABLE 10-1 INTERRUPT PRICRITY LEVEL/REGISTER SET SUMMARY

PSW BITS		CURRENT REGISTER SET	EXTERNAL INTERRUPT LEVEL ENABLED			
17	20		LEVEL 0	LEVEL 1	LEVEL 2	LEVEL 3
0	0	ANY SET	NO	NO	NO	NO
0	1	0	NO	NO	NO	NO
0	1	1	YES	NC	NO	NO
0	1	2	YES	YES	NO	NO
0	1	3	YES	YES	YES	NO
0	1	4	YES	YES	YES	YES
0	1	5	YES	YES	YES	YES
0	1	6	YES	YES	YES	YES
0	1	F	YES	YES	YES	YES
1	0	ANY SET	YES	YES	YES	YES
1	1	0	YES	NC	NO	NO
1	1	1	YES	YES	NO	NO
1	1	2	YES	YES	YES	NO
1	1	3	YES	YES	YES	YES
1	1	4	YES	YES	YES	YES
1	1	5	YES	YES	YES	YES
1	1	6	YES	YES	YES	YES
1	1	F	YES	YES	YES	YES

The processor recognizes I/C interrupts between the execution of instructions, or at the end of an iteration of an interruptible instruction. When an I/O interrupt is recognized, the following actions occur:

1. The current PSW is saved in registers 0 and 1 of the new set selected by the interrupt level. (PSW bits 0:31 are saved in register 0 and bits 32:63 in register 1.)

2. The PSW status word is loaded with the value Y'000028N0', where N specifies the new register set. This status enables higher level I/O interrupts and machine malfunction interrupts. Also note that memory address translation is disabled.

3. The I/O interrupt request is acknowledged and reset. The address of the interrupting device is placed in register 2 of the selected set. The status byte from the interrupting device replaces the contents of register 3. The device number and status are placed in the least significant bit positions in the register; the most significant bits are forced to zero. The four least significant bits of the status of the interrupting device are placed in the condition code.

4. The device number is added twice to X'0000D0' (the start of the interrupt service pointer table) to obtain the address within the table that corresponds to the interrupting device. The contents of this halfword of memory are fetched and examined to see if the interrupt is to be treated as an immediate interrupt or as an auto driver channel operation. If bit 15 of the halfword is zero, an immediate interrupt is required. If bit 15 of the halfword is one (the halfword is odd), an auto driver channel operation is required. If the interrupt is an immediate interrupt, the value in the table becomes the location counter portion of the current PSW. If the interrupt is an auto driver channel operation, then the least significant bit of the halfword is replaced by zero and the resulting value is placed in register 4 of the selected set. The auto driver channel is then activated.

10.5.6 Simulated Interrupt

The simulated interrupt results from executing a Simulate Interrupt (SINT) instruction when PSW bit 23 is zero. SINT is a privileged instruction, and may not be executed when PSW bit 23 is set.

Execution of the SINT instruction causes the processor to simulate acknowledgement of an enabled I/O interrupt request from an external device. The device address and interrupt level for the simulated interrupt are specified by the operands of the SINT instruction.

The states of PSW bits 17 and 20, normally used to enable and disable the various I/O interrupt levels, are ignored by the SINT instruction. For purposes of the simulated interrupt, I/O interrupts at all priority levels are assumed to be enabled. No pending device interrupt request is actually acknowledged by the processor as a result of executing the SINT instruction. With the exception of the differences described here, the simulated interrupt request is handled as detailed in Section 10.5.5.

CAUTION

DUE TO THE FACT THAT THE SINT INSTRUCTION IGNORES THE STATES OF PSW BITS 17 AND 20, IT SHOULD BE USED CAREFULLY BY PROGRAMS WHICH RUN IN REGISTER SETS 0, 1, 2, OR 3. FOR EXAMPLE, IF A PROGRAM EXECUTING IN REGISTER SET 2 ENABLES ONLY HIGHER-LEVEL INTERRUPTS, DATA IN THE REGISTERS OF SET 2 ARE NOT NORMALLY SUBJECT TO CHANGE AS A RESULT OF AN I/O INTERRUPT. HOWEVER, IF THE PROGRAM EXECUTING IN REGISTER SET 2 DOES A SINT CAUSING INTERRUPT LEVEL 3 (AND REGISTER SET 3) TO BE SELECTED, THE NEW PSW LOADED BY THE PROCESSOR CAUSES INTERRUPTS AT LEVELS 2, 1, AND 0 TO BE ENABLED. IF AN I/O INTERRUPT REQUEST AT LEVEL 2 OCCURRED, IT WOULD BE HONORED, CAUSING REGISTERS 0, 1, 2, AND 3 (AND PERHAPS 4) OF SET 2 TO BE OVERWRITTEN.

IF THESE REGISTERS ARE NOT STORED BEFORE THE SINT INSTRUCTION IS EXECUTED, DATA IN THE REGISTERS IS LOST, AND SYSTEM SOFTWARE COULD BE LEFT IN AN INDETERMINATE STATE.

The simulated interrupt is a software interrupt.

10.5.7 System Queue Service (SQS) Interrupt

When any of the instructions listed below is executed, as the instruction completes, bit 22 of the new PSW loaded by the instruction is tested. If the bit is zero, the SQS interrupt is disabled, and program execution continues according to the new PSW loaded.

MNEMONIC	MEANING
EPSR	Exchange Program Status Register
LDPS	Load Process State
LPSW	Load Program Status Word
LPSWR	Load Program Status Word Register

If bit 22 of the new PSW loaded by any of these instructions is set, the state of the system queue (whose physical address is found at physical location X'000080') is tested. The system queue is assumed to be maintained according to the circular list format. The number used field is fetched from the list header. If this field contains zero, the system queue is assumed to be empty, and program execution continues according to the new PSW loaded.

If the number used field for the system queue is not zero when it is tested, the following actions are taken to cause an SQS interrupt:

1. The current PSW, which was loaded by execution of one of the listed instructions, is stored in registers 14 and 15 of the set selected by the SQS interrupt new PSW found in memory at physical address X'000088'.

2. Register 13 of the selected set is loaded with the address of the system queue.

3. The SQS interrupt new PSW becomes the current PSW.

If the SQS interrupt occurs as a result of executing an EPSR instruction, the old PSW location counter presented to the interrupt service routine in register 15 points to the instruction following the EPSR instruction. If the interrupt occurs as a result of executing any of the other listed instructions, the old PSW location counter contains the value loaded by the instruction causing the interrupt.

Items may be added to the system queue while the SQS interrupt is enabled or disabled. The Add to Top of List (ATL) and Add to Bottom of List (ABL) instructions are normally used for this purpose. The fact that the items have been added to the system queue is recorded in the list header. Only when a new PSW is loaded which enables the SQS interrupt, is the state of the queue tested, and an interrupt allowed.

The system queue has a maximum size, as determined by the list header established by system software. If an attempt is made to add an item to the queue when it is already full, the data may be lost. This could result in system software being left in an indeterminate state.

Note that the address of the system queue contained in the system queue pointer must be aligned to a fullword boundary.

See Section 10.6 on Status Switching Instructions for a description of the EPSR, LDPS, LPSW, and LPSWR instructions.

The SQS interrupt is a deferred synchronous software interrupt.

10.5.8 Supervisor Call (SVC) Interrupt

The Supervisor Call (SVC) interrupt occurs when the SVC instruction is executed. This instruction and the resulting interrupt provide a means for any program to communicate with system software.

When the SVC instruction is executed, the processor takes the following actions:

1. The current PSW is saved in registers 14 and 15 of the set selected by the SVC interrupt new PSW found in memory at physical address X'000098'.

2. Register 13 of the selected set is loaded with the effective second operand address calculated for the SVC instruction executed. This is normally the address of an SVC parameter block, aligned to a fullword boundary.

3. The SVC interrupt new PSW becomes the current PSW, with a new location counter value chosen from the ordered list of halfwords at physical location X'9C'.

The old PSW location counter presented to the interrupt service routine in register 15 points to the instruction following the SVC instruction.

The SVC interrupt is a software interrupt and cannot be disabled.

10.5.9 System Breakpoint Interrupt

A system breakpoint results if a Breakpoint (BRK) instruction is executed when PSW bit 23 is zero. BRK is a privileged instruction, and may not be executed when PSW bit 23 is set.

Execution of the BRK instruction causes the processor to enter the console mode. In this mode, the processor is dedicated to communication with the system console terminal. Various registers and memory locaticns may be examined or modified by the user from the system console terminal while in this mode.

When the BRK instruction is executed, no registers or memory locations are mcdified. The PSW status and location counter are not modified by the BRK instruction. The location counter, at entry to the console mode, points to the BRK instruction.

When the run mode is entered from the console mode, PSW bit 16 is forced to zero, so that an instruction is fetched and executed. If the run mode is entered immediately after a BRK instruction is executed, the same BRK instruction results in another system breakpoint.

The system breakpoint interrupt is a software interrupt.

10.5.10 Arithmetic Fault Interrupt

The arithmetic fault interrupt results from either a fixed-point or a floating-point arithmetic operation, when the magnitude of the result is too large to be represented within the required number of bits. Division by zero is a special case and always results in an arithmetic fault interrupt. Interrupts for any of these reasons cannot be disabled.

Floating-point underflow occurs when the normalized result of a floating-point load, conversion, or other arithmetic operation is not zero, but is so small that it cannot be represented within the floating-point number system defined for the processor.

If PSW bit 19 is zero when floating-point underflow occurs, no arithmetic fault interrupt results. In this case, the result of the operation is set to "true zero". This means that every bit of the result is forced to zerc as the result is copied to its destination. If PSW bit 19 is set when floating-point underflow occurs, an arithmetic fault interrupt does occur.

When an arithmetic fault interrupt occurs, the following actions are taken:

1. The instruction causing the interrupt is aborted before the data in any register or memory location is modified.

2. The current PSW is stored in registers 14 and 15 of the set selected by the arithmetic fault interrupt new PSW found in memory at physical address X'000048'.

3. Register 13 of the selected set is loaded with a code to indicate the reason for the interrupt:

CODE	REASON FOR INTERRUPT
0	Fixed-point division by zero
1	Fixed-point quotient overflow
2	Floating-point division by zero
3	Floating-point exponent underflow
4	Floating-point exponent overflow

4. Register 12 of the selected set is loaded with the address of the instruction following the instruction causing the interrupt.

5. The arithmetic fault interrupt new PSW becomes the current PSW.

The old PSW location counter presented to the interrupt service routine in register 15 points to the instruction that caused the interrupt.

10.6 STATUS SWITCHING INSTRUCTIONS

Status switching instructions provide for software control of the system's interrupt structure. They also allow user-level programs to communicate efficiently with control software. All status switching instructions, except the supervisor call instruction, are privileged operations; therefore, all interrupt handling routines must run in the supervisor mode.

The status switching instructions described in this section are:

10.6.1	LPSW	Load Program Status Word
10.6.2	LPSWR	Load Program Status Word Register
10.6.3	EPSR	Exchange Program Status Register
10.6.4	SINT	Simulate Interrupt
10.6.5	SVC	Supervisor Call
10.6.6	BRK	System Breakpoint
10.6.7	PSF	Privileged System Function

10.6.1 Load Program Status Word (LPSW)

Assembler Notation		Op-Code	Format
LPSW	D2(X2)	C2	RX1,RX2
LPSW	A2(FX2,SX2)	C2	RX3

Operation

The 64-bit second operand replaces the current PSW.

Condition Code

Determined by the new PSW (bits 28:31).

Programming Notes

The R1 field of this instruction must be zero.

The second operand must be aligned to a fullword boundary.

This instruction is a privileged operation.

This instruction may be used to change register sets. The new set becomes active for execution of the next instruction.

If bit 22 of the new PSW is set, the state of the system queue is tested. If the queue is nonempty, a System Queue Service (SQS) interrupt occurs. In this case, the newly loaded PSW is saved as the old PSW when the SQS interrupt occurs.

10.6.2 Load Program Status Word Register (LPSWR)

Assembler Notation	Op-Code	Format
LPSWR R2R2	18	RR

Operation

The contents of the register specified by R2 replace bits 0:31 of the current PSW. The contents of the register specified by R2+1 replace bits 32:63 of the current PSW.

Condition Code

Determined by the new PSW (bits 28:31).

Programming Notes

The R1 field of this instruction must be zero.

The R2 field of this instruction must specify an even-numbered register.

This instruction may be used to change register sets. The new set becomes active for execution of the next instruction.

This instruction is a privileged operation.

If bit 22 of the new PSW is set, the state of the system queue is tested. If the queue is nonempty, a System Queue Service (SQS) interrupt occurs. In this case, the newly loaded PSW is saved as the old PSW when the SQS interrupt occurs.

10.6.3 Exchange Program Status Register (EPSR)

Assembler Notation	Op-Code	Format
EPSR R1,R2	95	RR

Operation

Bits 0:31 of the current PSW replace the contents of the register specified by R1. The contents of the register specified by R2 then replace bits 0:31 of the current PSW.

Condition Code

Determined by the new PSW (bits 28:31).

Programming Notes

R1 and R2 may specify any general-purpose registers.

If R1 and R2 specify the same register, bits 0:31 of the current PSW are copied into the register specified by R2, but otherwise remain unchanged.

This instruction may be used to change register sets. The new set becomes active for execution of the next instruction.

This instruction is a privileged operation.

If bit 22 of the new PSW is set, the state of the system queue is tested. If the queue is nonempty, a System Queue Service (SQS) interrupt occurs. In this case, the newly loaded PSW is saved as the old PSW when the SQS interrupt occurs.

10.6.4 Simulate Interrupt (SINT)

Assembler Notation		Op-Code	Format
SINT	I2(X2)	E2	RI1
SINT	R1,I2(X2)	E2	RI1

Operation

The least significant 10 bits of the second operand are presented to the interrupt handler as a device number. The device number is used to index into the interrupt service pointer table, simulating an interrupt request from an external device. The result is either an immediate interrupt or an auto driver channel operation.

Condition Code

Determined by the status of the addressed device, in the case of the immediate interrupt, or set by the auto driver channel at termination.

Programming Notes

If the R1 field of this instruction is not specified or contains zero, it is assumed that an interrupt from level 0 is required, and register set 0 is selected.

If the R1 field of the instruction is nonzero, the least significant 4 bits of the register specified by R1 designate the new register set, and consequently the new interrrupt level.

This instruction is a privileged operation.

This instruction causes the processor to load registers 0 through 3, or 0 through 4, of the new set as for a real interrupt request.

During the execution of this instruction, the device is addressed and the status byte is returned in register 3 of the new set.

If the specified device does not respond to the status request, register 3 of the new set contains X'00000004' due to time-out. If an immediate interrupt is being simulated, the V flag is also set in the condition code as a result of the time-out.

The SINT instruction does not cause any pending interrupt to be acknowledged.

10.6.5 Supervisor Call (SVC)

Assembler Notation		Op-Code	Format
SVC	N,D2(X2)	E1	RX1, RX2
SVC	N,A2(FX2,SX2)	E1	RX3

Operation

The second operand (normally the program address of an SVC parameter block) replaces bits 8:31 of register 13 of the set designated by the supervisor call new PSW status. Bits 0:7 of this register are forced to zero. The current PSW replaces the contents of registers 14 and 15 of that set. The fullword quantity located at X'000098' in memory replaces bits 0:31 of the current PSW. The 4-bit N field is doubled and added with X'00009C'. The halfword quantity located at the resultant address becomes the current location counter.

Condition Code

Determined by the new PSW (bits 28:31).

Programming Note

This instruction provides a means to switch from the protect mode to the supervisor mode. It is used by a program running under an operating system to initiate certain functions in the supervisor program. The second operand address is normally a pointer to the memory location of parameters needed by the supervisor program to perform the specified function. Such a pointer must indicate a parameter block aligned to a fullword boundary. The type of supervisor call is specified in the N field of the instruction. Sixteen different calls are provided for. Return from the supervisor is made by executing an LPSWR instruction specifying the stored old PSW in registers 14 and 15 of the set selected by the Supervisor Call interrupt new PSW (LPSWR R14).

10.6.6 System Breakpoint (BRK)

Assembler Notation	Op-Code	Format
BRK	88	SF

Operation

The BRK instruction causes the processor to enter the console mode.

Programming Notes

The location counter is not incremented.

This instruction is a privileged instruction.

10.6.7 Privileged System Function (PSF)

Assembler Notation	Op-Code	Format
PSF N,D2(X2)	DF	RX1,RX2
PSF N,A2(FX2,SX2)	DF	RX3

Operation

The PSF instruction may perform any one of 16 functions, as specified by the value contained in the N field. The assembler recognizes extended mnemonics which cause the proper value to be specified in the N field of this instruction. The nature of the specified function may vary from processor to processor. The following paragraphs detail PSF operations performed by this processor.

VALUE OF N	EXTENDED PSF MNEMONIC	MEANING
0	REL	Read Error Logger
1	LPSTD	Load Process Segment Table Descriptor
2	LSSTD	Load Shared Segment Table Descriptor
3	STPS	Store Process State
4	LDPS	Load Process State
5	ISSV	Save Interruptible State
6	ISRST	Restore Interruptible State
7	XSTB	Store Byte, no ECC
8	RMVF	Reset Memory Voltage Failure

Programming Notes

This instruction is a privileged instruction.

PSF functions selected by values of N other than those listed above are undefined for this processor and result in an illegal instruction interrupt.

10.6.7.1 Read Error Logger (REL)

Assembler Notation	Op-Code	Format
REL R2	DF0	RX1 (see programming notes)

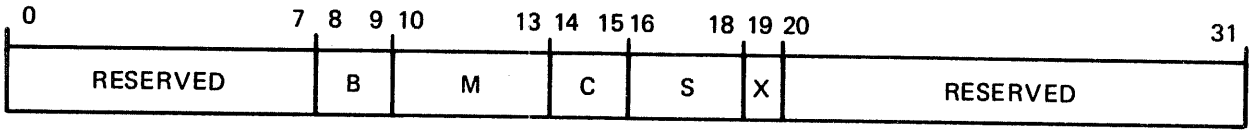
Operation

The register specified by R2 contains an error logger address as shown below. If bit 19 of the specified error logger address is zero, the 16 error logger bits at this address are read. These bits correspond to memory data errors detected since the error logger bits at that address were last read.

If bit 19 of the error logger address is set, the error logger status register is read. Bit 16 of this data is one if errors were detected since the status was last read; otherwise, bit 16 of this data is zero.

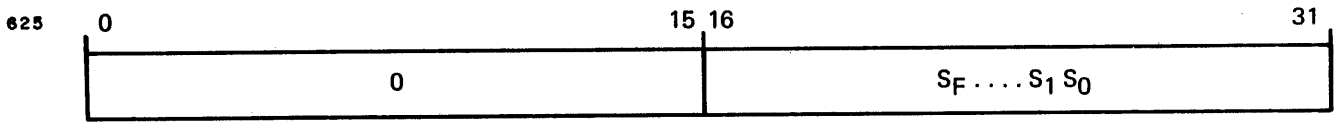
Data read from the error logger replaces the contents of the register specified by R2+1. Bit 16 of the data is copied to the L flag in the condition code. Once the data has been read from the error logger location, data at that location is set to zero.

624



BITS	MNEMONIC	USE
00:07	Reserved	Must be zero.
08:13	M	Selects one 256 kb memory array.
14:15	C	Selects one of the four columns within the memory array.
16:18	S	A syndrome code, modulo 8. The 16 error logger bits at a corresponding syndrome address are read if the X bit is zero.
19	X	If this bit is set in the error logger address, error logger status is read. The C and S fields are ignored in this case.
20:31	Reserved	Must be zero.

The format of the data read from the error logger is shown below. S(n) is the bit in the error logger at the address corresponding to the specified syndrome code plus n.



There may be multiple error loggers in the system, each uniquely accessible via the M field of the REL error logger address. The following paragraphs describe the relationship between an area of memory and the associated error logger.

There is an error logger for each memory bank. The bank also consists of a bank controller and at least one main memory storage module (STM). For this processor, there is only one bank within a given memory block. The memory block is noninterleaved.

The lowest-addressed byte within a memory block is located on a 1-megabyte boundary. There may be more than one memory block within the system. The primary (lowest-addressed) memory block is labeled Block 0. Memory within the block is configurable in minimum increments of 256 kb. In a system with both local and shared memory, local memory occupies low address space. Local memory cannot occupy the same memory block as shared memory.

Condition Code (after reading error logger status)

C	V	G	L	
X	X	X	0	No new error bits in selected bank
X	X	X	1	At least one new error bit in selected bank

Programming Notes

This instruction is a privileged instruction.

The R2 field of this instruction must specify an even-numbered register.

Reading error logger status sets the error bit to zero, but does not necessarily zero the error logger bits at any syndrome address.

REL is assembled as an RX1 format instruction, in which the displacement field is always zero.

REL is an extended PSF mnemonic.

If the error logger is not equipped, undefined data is returned by this instruction.

Refer to the Processor Maintenance Manual for further details of error logger operation.

PROGRAMMING EXAMPLE: REL

In order to determine the number and location of error loggers in the system, it is first necessary to zero the error bit in each possible error logger status register. The following sequence requires that location MEMTOP contain the address of the last byte in memory. For the example, MEMTOP contains Y'003FFFFF' for a 4 Mb memory configuration.

Location	Contains	Assembler Notation			Comments
1000	C820 0010		LHI	R2,X'10'	BANK INCREMENT
1004	C800 1000		LHI	R0,X'1000'	X BIT, LOGGER 0 BANK 0
1008	DF00 0000	LAB1	REL	R0	READ BANK 0 STATUS REG
100C	0A02		AR	R0,R2	ADVANCE TO BANK 1
100E	DF00 0000		REL	R0	READ BANK 1 STATUS REG
1012	0A02		AR	R0,R2	ADVANCE TO BANK 2
1014	DF00 0000		REL	R0	READ BANK 2 STATUS REG
1018	0A02		AR	R0,R2	ADVANCE TO BANK 3
101A	DF00 0000		REL	RD	READ BANK 3 STATUS REG
101E	FA00 000F FFD0		AI	R0,Y'FFFD0'	ADVANCE TO NEXT 1 MB BOUNDARY, @ BANK 0
1024	5500 1090		CL	R0,MEMTOP	PAST TOP OF PHYS MEMORY ?
1028	4280 1008		BL	LAB1	BRANCH: NO

At this time, the error logger error status bits are zero for all loggers associated with memcry from address X'000000' through the address value contained in MEMTOP.

The following example shows how to create a new error indication in an error logger. On entry, R6 contains the running PSW, with bit 18 forced to zero. This PSW is used to allow errors to be logged without causing a machine malfunction interrupt.

Location	Contains	Assembler Notation		Comments
1200	5821 0000	L	R2,0(R1)	GET MEMORY DATA FULLWORD
1204	0802	LR	R0,R2	COPY FOR XSTB
1206	C700 0001	XHI	R0,1	CHANGE LS BIT
120A	DF71 0003	XSTB	3(R1)	STORE LS BYTE OF FULLWORD WITH SINGLE-BIT ERROR
120E	9556	EPSR	R5,R6	ZERO PSW BIT 18 (MMF)
1210	5831 0000	L	R3,0(R1)	READ,CREATE ERROR
1214	9565	EPSR	R6,R5	RESTORE ENTRY PSW
1216	5020 0000	ST	R2,0(R1)	RESTORE MEMORY DATA

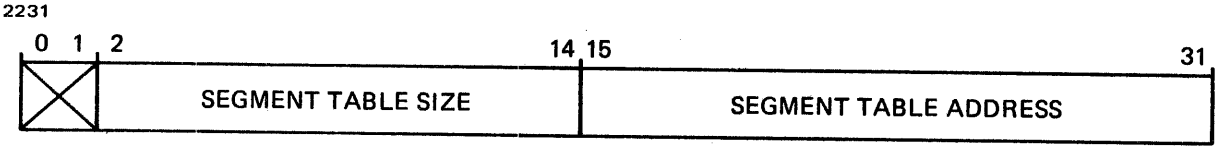
If the preceding sequence is used with a program address corresponding to Bank 0 of the memory block under test, and if the error is shown also in Bank 3, the block is not interleaved. If the error is not shown in Bank 3, but is shown in Bank 1, then the block is 2-way interleaved; otherwise, the block is 4-way interleaved. Note that once the error logger status is read and the error is shown to have been detected, the status bit is set to zero. This requires that the error be re-created before continuing the sequence. A similar technique may be used to determine the amount of memory in each bank of an error logger. The error is created at the lowest 1 Mb boundary in the bank. The next 1 Mb boundary of the bank is tested for the error. If the error bit is set, that address space belongs to the error logger being interrogated. When the error disappears, a new memory block may have been entered. Note that "address wrap" may occur if an address larger than the configured memory size is used. This may be detected by interrogating the logger at the same bank of Block 0.

10.6.7.2 Load Process Segment Table Descriptor (LPSTD)

Assembler Notation		Op-Code	Format
LPSTD	D2(X2)	DF1	RX1,RX2
LPSTD	A2(FX2,SX2)	DF1	RX3

Operation

The second operand address points to a fullword Process Segment Table Descriptor (PSTD), which has the following format:



Bits 0:1 Are reserved, and must be zero.

Bits 2:14 Contain the number of doubleword entries in the Process Segment Table, minus one. This number defines the Maximum Valid Program Address (MVPA) for the executing program (process).

Bits 15:31 Contain the absolute address of the Process Segment Table in main memory, divided by 128.

The data in the Process Segment Table is used in translation of program addresses from program to physical address space when PSW bit 21 is set.

Condition Code

Unchanged

Programming Notes

The operand address must be aligned to a fullword boundary.

The LPSTD instruction may be executed regardless of the state of PSW bit 21.

The new Process Segment Table is available for execution of the next instruction which is executed with PSW bit 21 set.

This instruction is a privileged instruction.

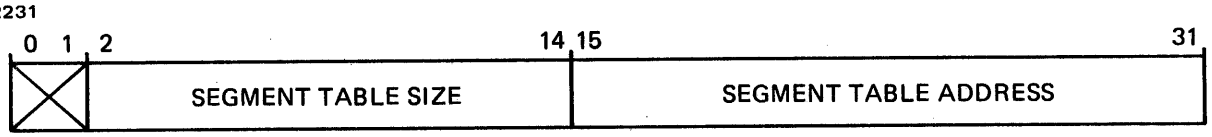
LPSTD is an extended PSF mnemonic.

10.6.7.3 Load Shared Segment Table Descriptor (LSSTD)

Assembler Notation		Op-Code	Format
LSSTD	D2(X2)	DF2	RX1,RX2
LSSTD	A2(FX2,SX2)	DF2	RX3

Operation

The second operand address points to a fullword Shared Segment Table Descriptor (SSTD), which has the following format:



Bits 0:1 Are reserved, and must be zero.

Bits 2:14 Contain the number of doubleword entries in the Shared Segment Table, minus one.

Bits 15:31 Contain the absolute address of the Shared Segment Table in main memory, divided by 128.

The data in the Shared Segment Table is used in translation of program addresses from program to physical address space when PSW bit 21 is set, if the Process Segment Table entry specifies that the segment is shared.

Condition Code

Unchanged

Programming Notes

The operand address must be aligned to a fullword boundary.

The LSSTD instruction may be executed regardless of the state of PSW bit 21.

The new Shared Segment Table is available for execution of the next instruction which is executed with PSW bit 21 set.

Following an LSSTD instruction, the Process Segment Table Descriptor must be loaded, using the LPSTD or LDPS instruction, before attempting MAT translation with the newly defined shared segment table.

This instruction is a privileged instruction.

LSSTD is an extended PSF mnemonic.

10.6.7.4 Store Process State (STPS)

Assembler Notation	Op-Code	Format
STPS D2(X2)	DF3	RX1,RX2
STPS A2(FX2,SX2)	DF3	RX3

Operation

The process state, defined by the old PSW in registers 14 and 15 of the current set, is saved in the area of memory whose starting address is specified by the operand. The area has the following format:

NORMAL OFFSET (BYTES)	STORED DATA
0-7	Process PSW
8-11	Reserved - not used
12-75	Process general registers
76-139	Process interruptible state
140-235	Single- and double-precision floating-point registers

Condition Code

Unchanged

Programming Notes

The operand address must be aligned to a fullword boundary.

This instruction is a privileged instruction.

STPS is an extended PSF mnemonic.

The process general register set is specified by the old PSW in register 14 when this instruction is executed.

If bit 14 of the process PSW in register 14 is zero, the process interruptible state is not saved, and the save area is compacted accordingly. In this case, the process' floating-point registers are saved beginning at an offset of 76 bytes from the specified operand address.

If bit 13 of the process PSW in register 14 is set, or if the processor is not equipped with floating-point registers, then floating-point registers are not saved, and the save area is compacted accordingly.

10.6.7.5 Load Process State (LDPS)

Assembler Notation		Op-Code	Format
LDPS	D2(X2)	DF4	RX1,RX2
LDPS	A2(FX2,SX2)	DF4	RX3

Operation

Data from the area of memory specified by the operand replaces the current process state. The area has the following format:

NORMAL OFFSET (BYTES)	STORED DATA
0-7	Process PSW
8-11	Process segment table descriptor
12-75	Process general registers
76-139	Process interruptible state (if bit 14 in saved PSW is set)
140-235	Process single-precision and double-precision floating-point registers (if bit 13 in saved PSW is zero)

The new PSW at the operand address specifies the general register set which is loaded from the save area. If bit 14 of the new PSW is set, the interruptible state is loaded from the save area. If bit 13 of the new PSW is zero, and the processor is equipped with floating-point registers, then the single- and double-precision floating-point registers are loaded from the save area. If bit 21 of the new PSW is set, the Process Segment Table Descriptor is loaded. Finally, the new PSW at the operand address becomes the current PSW.

Programming Notes

The operand address must be aligned to a fullword boundary.

This instruction is a privileged instruction.

LDPS is an extended PSF mnemonic.

If bit 14 of the new PSW is zero, the process interruptible state is not loaded, and the save area is assumed to be compacted accordingly. In this case, the process' floating-point registers are loaded from memory beginning at an offset of 76 bytes from the specified operand address.

If bit 13 of the new PSW is set, or if the processor is not equipped with floating-point registers, the process' floating-point registers are not loaded, and the save area is assumed to be compacted accordingly.

If bit 22 of the new PSW is set, the state of the system queue is tested before testing the wait bit (bit 16). If the queue is nonempty, a System Queue Service (SQS) interrupt occurs. In this case, the newlyloaded PSW is saved as the old PSW when the SQS interrupt occurs.

The state of the wait bit (PSW bit 16) is tested before the next instruction is executed.

The process register set is selected in order to load the process general registers. All data is fetched from the save area before the process PSW is loaded. If a fault occurs during the execution of this instruction, one or more of the specified registers may have been modified. The old PSW presented to the fault interrupt service routine in register 14 may select the general register set specified by the process PSW in the save area, but is otherwise the same as the PSW in effect when this instruction is fetched and executed. The old PSW location counter presented to the interrupt service routine in register 15 points to the LDPS instruction.

10.6.7.6 Save Interruptible State (ISSV)

Assembler Notation		Op-Code	Format
ISSV	D2(X2)	DF5	RX1,RX2
ISSV	A2(FX2,SX2)	DF5	RX3

Operation

The contents of the interruptible instruction scratchpad registers are stored in the 16 fullwords of memory starting at the address specified by the operand.

Condition Code

Unchanged

Programming Notes

The operand address must be aligned to a fullword boundary.

This instruction is a privileged instruction.

ISSV is an extended PSF mnemonic.

10.6.7.7 Restore Interruptible State (ISRST)

Assembler Notation		Op-Code	Format
ISRST	D2(X2)	DF6	RX1,RX2
ISRST	A2(FX2,SX2)	DF6	RX3

Operation

The interruptible instruction scratchpad registers are loaded from the 16 fullwords in memory starting at the address specified by the operand.

Condition Code

Unchanged

Programming Notes

The operand address must be aligned to a fullword boundary.

This instruction is a privileged instruction.

ISRST is an extended PSF mnemonic.

10.6.7.8 Store Byte, No ECC (XSTB)

Assembler Notation		Op-Code	Format
XSTB	D2(X2)	DF7	RX1,RX2
XSTB	A2(FX2,SX2)	DF7	RX3

Operation

The contents of bits 24:31 of general register 0 are stored in memory at the address specified by the operand, without changing the error correction code bits for the specified memory location.

Condition Code

Unchanged

Programming Notes

This instruction is a privileged instruction.

XSTB is an extended PSF mnemonic.

This instruction may be used in conjunction with the read error logger instruction to test the operation of the Error Correction Codes (ECC).

10.6.7.9 Reset Memory Voltage Failure (RMVF)

Assembler Notation	Op-Code	Format
RMVF	DF8	RX1 (See programming notes)

Operation

The processor's internal Memory Voltage Failure (MVF) indication is reset. The MVF indication is set only as a result of the voltages to main memory not being maintained within acceptable margins during a power fail/restore sequence.

Condition Code

Unchanged

Programming Notes

This instruction should be executed by software loaded via the optional LSU, after all interrupt new PSWs have been established. Proper use of this instruction prevents a potential runaway condition in the event of multiple power fail/restore sequences.

MVF is reset by the processcr when the console mode is entered.

This instruction is a privileged instruction.

RMVF is an extended PSF mnemonic.

RMVF generates an RX1 format instruction, in which the displacement field is always zero.

CHAPTER 11
MEMORY MANAGEMENT

11.1 INTRODUCTION

The Memory Address Translator (MAT) supports:

- 2^{24} byte (16 Mbyte) physical address space
- 2^{24} byte (16 Mbyte) program address space
- Segmentation
- Shared segments
- Read, write, and execute protection
- 4 levels of hardware controlled access to segments

Figure 11-1 provides a block diagram of the data structures required for various types of translation from program to physical address space.

Program address space consists of all memory addresses which may be specified by the executing program. Physical address space consists of all addresses resulting from translation of program addresses to addresses corresponding to actual memory locations. The Memory Address Translator (MAT) performs the translation as detailed in Figure 11-1 and in the following paragraphs.

The segment number field of the program address is used as an index into the process segment table to select a segment table entry. The segment table entry points either to a segment or the shared segment table. (See Figure 11-1.)

If the segment table entry points to the segment, then the offset selects the desired address relative to the beginning of the segment.

If the segment table entry points to a shared segment table, the value is used as an index relative to the beginning of the shared segment table where a segment table entry for the segment can be found. The segment table entry in the shared segment table cannot point to a shared segment table.

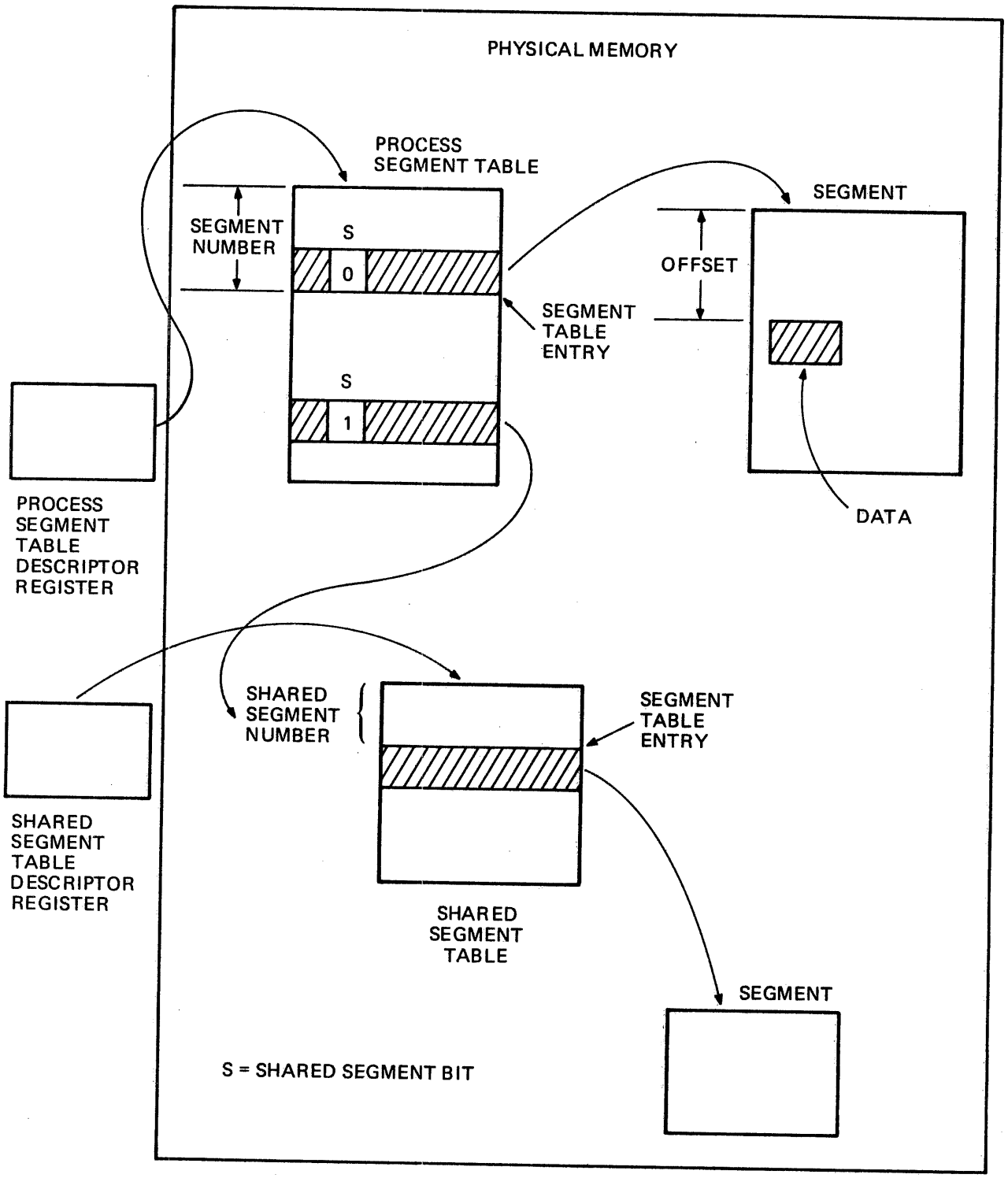


Figure 11-1 Memory Address Translation

11.2 ADDRESS SPACE

This processor supports management of a 2^{24} byte physical or program address space. When physical or program addresses are manipulated, they are treated as 24-bit quantities. In general, 32-bit quantities are available to the processor for address calculation. When intermediate calculations are complete, bits 0 through 7 of the 32-bit effective result are forced to zero or discarded, giving a calculated address 24 bits in length, which occupies bits 8 through 31 of the 32-bit effective result.

In some instances, an address consisting of less than 24 bits may be used by the processor. Such an address is extended to 24 bits in length by forcing the higher-order bits to zero.

11.2.1 Physical Address Space

The Memory Address Translator (MAT) is disabled when PSW bit 21 is zero. When the MAT is disabled, any of the 2^{24} byte maximum available memory may be directly accessed. In those cases where less than 2^{24} bytes of memory are configured, a machine malfunction fault condition is likely to occur as a result of attempting to access memory outside the available limits.

11.2.2 Program Address Space

The Memory Address Translator (MAT) is disabled when PSW bit 21 is zero. When enabled or disabled, the MAT may be programmed so that when translation is enabled, it is possible for a program to run in a maximum program address space of 2^{24} bytes. Program addresses generated during the execution of such a program are translated to physical addresses used in accessing memory by the MAT.

If a program address space of less than 2^{24} bytes has been created, and a program address is generated which is outside the limits of the program address space, a Memory Address Translator fault occurs.

The MAT, when properly programmed, allows simultaneous execution of concurrent processes while protecting each process from interference by the other processes in the system. Violation of any of the enabled protection mechanisms causes a Memory Address Translator fault to occur. A description of such faults may be found in the section on Memory Address Translator Faults in this chapter.

If a physical address space of less than 2^{24} bytes exists, and program address translation by the MAT results in a physical address which is outside the limits of physical address space, a machine malfunction fault ccndition is likely to occur. Proper programming of the MAT causes a program address which would result in such a physical address to be intercepted before reaching the memory system.

Figure 11-2 shows a 24-bit program address as it would be contained in a 32-bit general register or memory location. The program address is comprised of two fields: SEGMENT and OFFSET. The significance of each field is described in the following paragraphs.

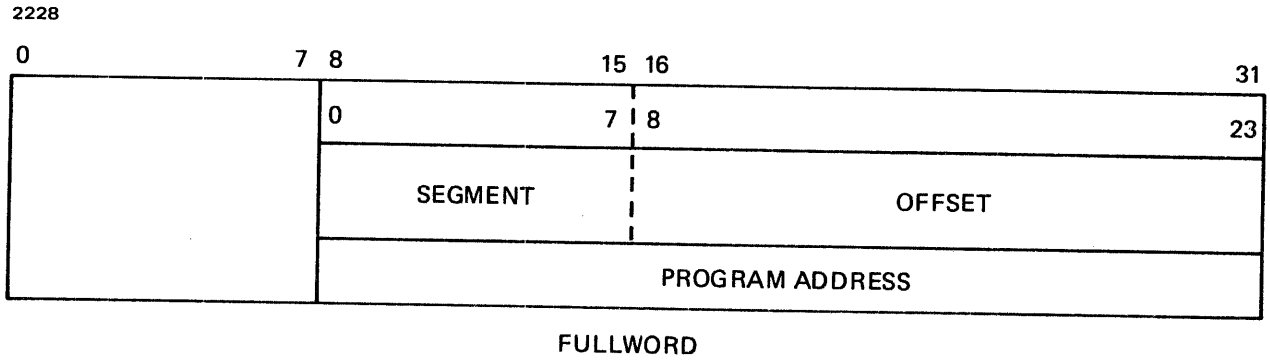


Figure 11-2 Program Address

11.2.2.1 Segment Field

The maximum program address space of 2^{24} bytes is divided into 2^{8} (256) segments of 2^{16} (65536) bytes each. A particular 64 kbyte segment is selected by the most significant 8 bits of the program address (this is shown graphically in Figure 11-2 as bits 0 through 7 of the program address, or bits 8 through 15 of the fullword containing the address). For example, program addresses in the range Y'000000' - Y'00FFFF' select segment 0, as the most significant 8 bits of each address are zero. Program addresses in the range Y'4F0000' - Y'4FFFFF' select segment $4F_{16}$ (79_{10}), as the most significant 8 bits of the address are $4F_{16}$.

11.2.2.2 Offset Field

The offset field is comprised of the least significant 16 bits of the program address (this is shown graphically in Figure 11-2 as bits 8 through 23 of the program address, or bits 16 through 31 of the fullword containing the address). The value contained in this field is used as a byte offset into the selected segment. For example, program address Y'09F001' specifies byte F001 of segment 9. The offset field of the program address has no special significance to the MAT, except with respect to segment limit checking (see Section 11.3.3.2).

11.2.3 Selection of Program or Physical Addressing

PSW bit 21, the relocation/protection bit, controls selection of program or physical addressing. When bit 21 is zero, the Memory Address Translator (MAT) is disabled. In this mode, all addresses generated are physical addresses. No segment table is used; hence, no level checking, access mode checking, etc., is performed. Bits 10 and 11 of the PSW (the access level bits) are ignored in the physical mode.

The user of the physical mode must take care when modifying memory. The fact that a data area has been modified is not recorded by hardware. If it is desired to reflect the modification information in the segment tables, this must be done explicitly by the program running in the physical mode.

When PSW bit 21 is set, the MAT is enabled. All addresses generated are program addresses which are translated to physical addresses using the segment tables. System software must ensure that segment table addresses have been specified via the LPSTD and LSSTD instructions (as in the section on Setting of the Program Address Space Size in this chapter) before the MAT is enabled.

When the MAT is enabled, bits 10 and 11 of the PSW indicate the level at which the program is running. When a program address is generated, the access level specified in the STE is compared against the contents of bits 10 and 11 of the PSW. If the contents of bits 10 and 11 are greater than or equal to the access level specified in the STE, then access to the segment is permitted; otherwise, a MAT fault occurs. System software should set bits 10 and 11 of the PSW according to the level at which the process is running to insure protection of segments.

11.3 TRANSLATION FROM PROGRAM TO PHYSICAL ADDRESS SPACE

The mapping of program address space to physical address space is accomplished using information supplied in a segment table. A segment table contains one doubleword entry, called a Segment Table Entry (STE), per segment in the program address space created. The segment number field of the program address is used as an index into the segment table for selection of an STE. A segment table must be aligned to a 2^7 (128) byte boundary in physical memory. The table may contain from 1 to 256 doubleword entries.

11.3.1 Shared and Private Segments

There may be a number of processes resident in the system at any given time. Each of these processes has its own program address space requirements, reflected in the process segment table associated with that process. Consequently, there may be several Process Segment Tables (PSTs) in memory concurrently, although only one, the segment table for the active process, may be known to the MAT at any given time. Section 11.3.2 contains details on how the active Process Segment Table is specified.

Segments of the program address space of a process which are used only by that process are called private segments. Other segments of the program address space may exist that are shared with other processes; these segments are consequently called shared segments. Although the STE describing a shared segment may be replicated in the segment tables associated with each process using the segment, it is preferable to maintain a separate Shared Segment Table (SST). For a shared segment, the process STE has an indication that the segment's description is not found in the Process Segment Table (PST), but is instead found in the Shared Segment Table. A detailed explanation of this is found in the section on Hardware Segment Table Entry.

The data contained in a segment must be stored in contiguous locations in physical memory. This is called Unpaged Allocation. For unpaged allocation, each segment must be aligned to a 2 (2048) byte boundary in physical memory.

11.3.2 Segment Table Descriptors and Their Use

The Memory Address Translator is enabled only when PSW bit 21 is set. Prior to enabling the MAT, the locations and sizes of the Process Segment Table and Shared Segment Table to be used must be identified to the system by loading the appropriate descriptor registers. These registers can be changed while MAT is enabled. To specify the address of the Process Segment Table to the system, a Load Process Segment Table Descriptor (LPSTD) instruction is used; to specify the address of the shared segment table, the Load Shared Segment Table Descriptor (LSSTD) instruction is used.

The STDs are volatile quantities. This means that in the event of a power fail, the values loaded must be assumed to have been destroyed. The power restore routine must reload the SSTD and PSTD before enabling the MAT.

11.3.2.1 Format of a Segment Table Descriptor

A Segment Table Descriptor (STD) is a fullword quantity, as shown in Figure 11-3.

Bits 0 and 1 of the STD are reserved and must always be zero. Bits 2 through 14 specify the segment table size, minus 1. For example, if the segment table size were 4, this field would have a value of 3. For a process STD (PSTD), this field has a maximum value of 255 (Y'FF'). For a shared STD (SSTD), this field has a maximum value of 8191 (Y'1FFF').

Bits 15 through 31 of the STD specify the segment table physical address, divided by 2^7 (128). A segment table must be aligned to a 2^7 byte boundary in physical memory. For example, if a segment table starts at location Y'035F80', then bits 15:31 of the STD contain Y'06BF' (Y'035F80'; divided by 2^7). The value of 0 for this field is a reserved value and, therefore, no segment table can start at physical address 0.

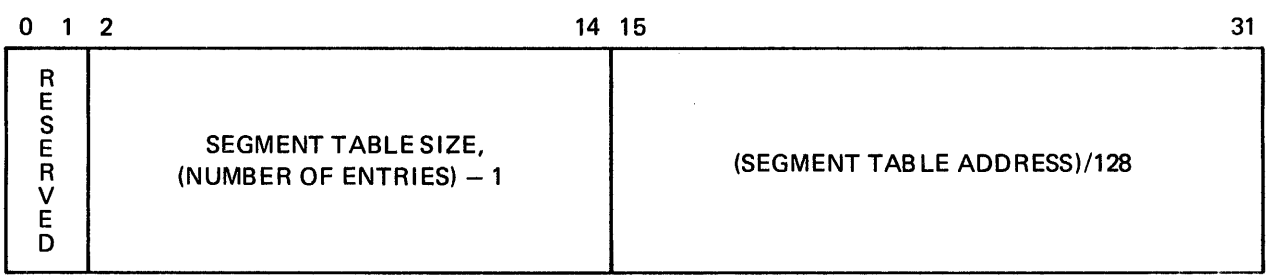


Figure 11-3 Segment Table Descriptor

11.3.2.2 Setting the Program Address Space Size

When a PSTD is loaded, its segment table size field determines the maximum valid program address. The maximum valid program address (MVPA) is defined by the following formula:

MVPA = (number of segment table entries)*(Y'10000') + X'FFFF'

The MVPA includes address space for the zeroth Segment Table Entry (STE).

For example, if the specified segment table size in the STD is 5, (requiring 6 segment table entries) then program addresses in the range Y'000000'-Y'05FFFF' are considered valid. If a program address is generated which is greater than the MVPA, a MAT fault occurs. (See Section 11.4 cn MAT Faults.)

Within the valid program address space, there may be segments which are not used. For example, a program address space of a process uses segments 0, 1, 2, and 5 while segments 3 and 4 are unused in that process. In this case, the segment table must contain entries for segments 3 and 4. To indicate that each of these segments is unused, its STE indicates that it is nonpresent and unused. (See Section 11.3.3.3 on Software Segment Table Entry).

11.3.3 Segment Table Entries

Figure 11-4 represents a segment table entry. Entries in the shared segment table and process segment table have the same format, with minor differences which are detailed in the following text.

11.3.3.1 Segment Table Entry Size

An STE is a 64-bit quantity. Bits 0:31 are the hardware STE and bits 32:63 are the software STE. The hardware STE contains the necessary information to enable hardware to perform program to physical address translation.

The software STE contains information used by system software to manage the process address space and keep track of segment history.

11.3.3.2 Hardware Segment Table Entry

The hardware portion of the STE (HSTE) is contained in bits 0:31 of the STE. The HSTE is comprised of several fields as shown in Figure 11-4. Reserved fields of the HSTE must always contain zero.

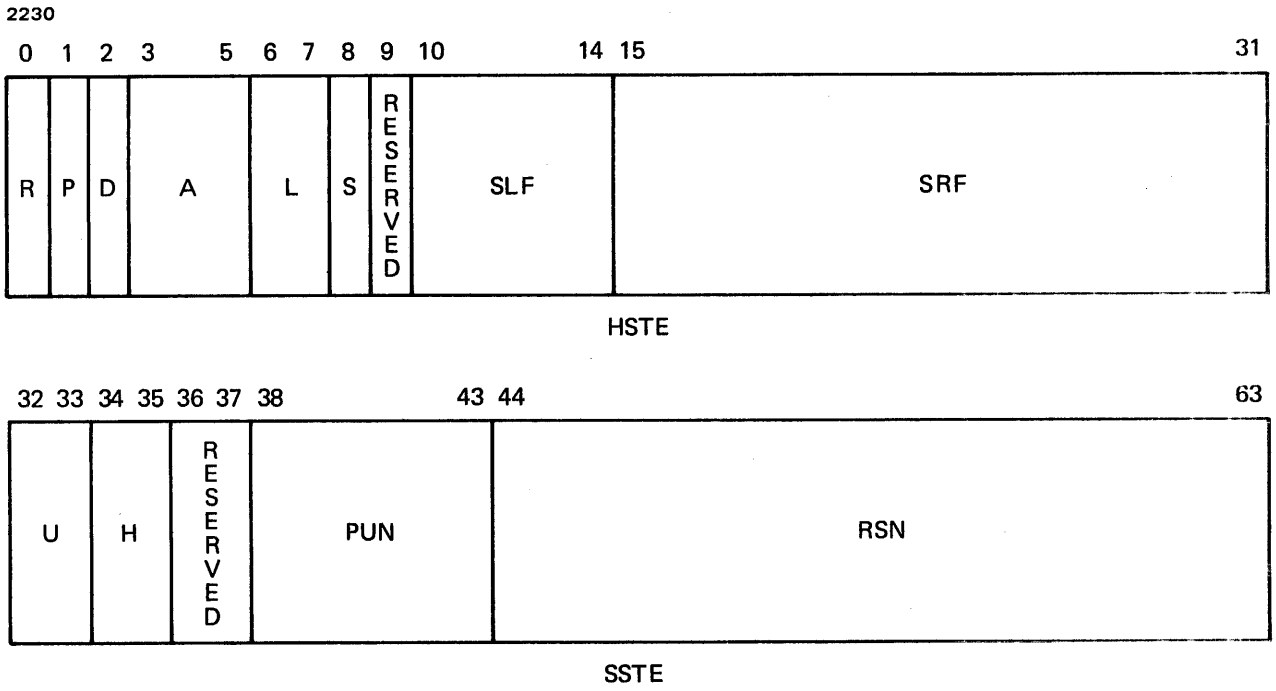


Figure 11-4 Segment Table Entry

R - Reference Bit - Bit 0 of the HSTE is called the reference (R) bit. This bit of the HSTE is set by the hardware when the segment is referenced by software. A segment is said to be referenced when an attempt is made to read, write, or execute the contents of the segment. DMA I/O does not set the reference bit for the participating segment.

P - Presence Bit - Bit 1 of the HSTE is the presence (P) bit. The presence bit is set when the segment described by the STE is in memory; it is zerc when the segment is not present in memory. A segment is said to be present when any portion of the segment is in memory. When P=0, hardware ignores the contents of the rest of the HSTE (Bits 2 through 31), but the R bit is set as a result of the attempted access. Referencing a segment that is not present (P=0) causes a fault. (See Section 11.4.1.2 on Nonpresence Fault.)

D - Dirty Bit - Bit 2 of the HSTE is called the dirty (D) bit. This bit of the HSTE is set by the hardware when a program modifies (writes into) a segment. This bit is not set by DMA I/O operations that modify the segment. If P=0, attempts to modify the segment do nct affect the dirty bit. If the PST STE indicates that the segment is shared, the D bit is set in both the SST STE, and in the PST STE, for a write.

A - Access Mode Bits - Bits 3 through 5 of the HSTE are called the access mode (A) bits. These bits determine the allowed modes of access to the segment. The three modes of access to a segment are: read access, write access, and execute access. If an attempt is made to access a segment in a manner not permitted by the setting of the access mode bits, a MAT fault occurs. For example, instructions cannot be fetched from a segment unless execute access is enabled. Section 11.4 contains a detailed definition of all MAT faults. Table 11-1 defines access field settings and the types of access that are enabled.

TABLE 11-1 SEGMENT ACCESS FIELD SETTINGS

SETTING	READ ENABLED	WRITE ENABLED	EXECUTE ENABLED
000	No	No	No
001	No	No	Yes
010	No	Yes	No
011	No	Yes	Yes
100	Yes	No	No
101	Yes	No	Yes
110	Yes	Yes	No
111	Yes	Yes	Yes

L - Access Level Bits - Bits 6-7 of the HSTE are called the access level (L) bits. The L field is used in conjunction with bits 10 and 11 of the Program Status Word (PSW) to determine if a program can access a segment. If the contents of PSW bits 10 and 11 are greater than or equal to the L bits, then access to the selected segment is allowed; if the value of PSW bits 10 and 11 is less than L, then a fault occurs. The L bits are checked before the A bits. See Section 11.4 for a detailed description of MAT faults.

S - Shared Segment Bit - Bit 8 of the HSTE is called the Shared Segment (S) bit. If this bit is zero, the MAT performs protection and relocation functions as defined for a private segment. The S bit must be zero for all STEs in the SST.

If the S bit is set in a Process Segment Table entry, the selected segment is a shared segment. In this case, the SRF field of the PST STE is used as a byte offset into the SST. The SST STE found at the resulting address is used by the MAT in performing protection and relocation functions, as follows:

The values of the A fields in the PST STE and the SST STE are ANDed to determine the allowed access modes. All other MAT protection and relocation functions are performed using data from the SST STE.

SLF - Segment Limit Field - Bits 10 through 14 of the HSTE are called the Segment Limit Field (SLF). The SLF is used to indicate the size of a segment. The SLF contents are determined according to the following formula:

$$SLF = (\text{size of segment}) \text{ divided by } (2^{11}) - 1$$

For example, for a segment of size 2^{11} bytes, the SLF would contain 0. When a program address is generated, the contents of bits 8:12 of the program address is compared to the SLF. If the SLF is less than this number, a MAT fault occurs (see Section 11.4).

The granularity of memory allocation is 2^{11} (2,048) bytes. This means that memory must be allocated in 2^{11} byte units. For example, if a segment requires Y'3542' bytes, 14 kbytes of memory must be allocated, though only part of the last 2,048 bytes is used.

SRF - Segment Relocation Field - Bits 15 through 31 of the HSTE are called the Segment Relocation Field (SRF). The interpretation of the SRF depends upon the setting of the S bit.

If S is set in the PST, the PST SRF field contains a byte offset into the SST at which the STE for the segment can be found. If the value contained in the PST SRF field is greater than the size in bytes of the SST, a MAT fault occurs. See Section 11.4 for a detailed description of MAT faults.

If S is zero in the PST, the SRF contains the physical address of the segment in memory, divided by 2^7. For example, if the segment starts at physical address Y'146800', the SRF field of the HSTE should contain X'28D0'.

Note that for a shared segment, the least significant 3 bits of the SRF field in a PST HSTE must be zero, as the indicated SST HSTE is aligned to a doubleword boundary. The least significant 4 bits must be zero for the SRF field in all other cases, as the SRF is the address of a segment aligned to a 2^{11} byte boundary, divided by 2^7. If the MAT attempts to use an SRF field which has a one in any of these trailing bits, the results are undefined.

11.3.3.3 Software Segment Table Entry

Bits 32-63 of the segment table entry are called the Software Segment Table Entry (SSTE). These bits are used by software in managing the program address space, and have no hardware significance to the processor.

The information presented in this section details one possible scheme for software management of memory. The fullword SSTE is available for any software memory management scheme. The OS/32 MT operating system software does not manage memory as suggested in this section.

a. Usage Mode Bits - Bits 0 and 1 of the SSTE (bits 32 and 33 of the STE) are called the usage mode (U) bits. In conjunction with the P bit in the HSTE, these bits indicate the state of each segment. Possible states of a segment are:

State	P	U
Unused	0	00
Used	0	01
Loading	0	10
Unloading	0	11
Active	1	00
I/O Ongoing	1	01
Unload Pending	1	10

1. Unused State - If a segment is logically an invalid portion of the program address space of a process, then it is said tc be in the unused state. For example, if a process has data in segments 0, 1, 2, and 5, but has no data in segments 3 and 4, references to segments 3 and 4 are invalid. Since segments 3 and 4 must each have an STE, the fact that these segments represent an invalid portion of the program address space of the process is indicated by setting P=0 and U=0 for the unused state. Since P=0 in the HSTE, any references to such a segment cause a MAT fault. The fault handler, using the U field, may determine that the fault was caused by a reference to an invalid portion of the process program address space and can take appropriate action.

2. Used State - If a segment is logically a valid part of the process program address space, but is not physically present in memory, that segment is said to be in the used state. When a reference is made to such a segment, a MAT fault occurs since P=0 in the HSTE. The fault handler, using the U field, may determine that the fault was caused by a reference to a used segment, and then take action to load the segment.

   When a used segment is to be loaded, the segment state is changed by scftware from used to loading as explained in the next paragraph.

3. Loading State - If a segment that is logically part of the program address space of the process is being moved from backing store into physical memory, it is said to be in the loading state. If a reference is made to a segment that is in the loading state, a fault occurs since P=0 in the HSTE. The fault handler normally places the process that made the reference into a wait state, pending completion of the loading operation.

   When a segment has successfully been loaded, software normally changes its state from loading to active. At this point, any process that faulted because it referenced the segment in the loading state, and was consequently put into a wait state, may resume execution.

4. Unloading State - When a segment that is logically part of the program address space of the process is being temporarily removed from physical memory and copied to back-up store, it is said to be in the unloading state. Any references to such a segment cause a MAT fault, because P=0 in the HSTE.

When a MAT fault occurs and the U field indicates that it was caused by a segment in an unloading state, the fault handler has a choice of actions it can take. For example, if the segment was being unloaded to make physical memory space available, the process that made the reference can be put into a wait state. When the unload completes, the physical memory that the segment occupies may be retained and the segment put into an active state. If it is desired to give the physical memory that the segment occupied to another segment, then the unloaded segment should be put into the used state and the fault should be treated in the same manner as faults that occur in a used state.

In some systems, a segment may be unloaded because it is being removed not only from physical memory but is also being removed logically from the program address space of the process. A dynamically attached and detached buffer segment is an example of such a segment. If a segment in an unloading state is being logically removed from the program address of the process, the segment goes into an unused state at the completion of the unloading operation. Faults generated by references to such segments are normally errors.

5. Active State - When a segment is logically in the program address space of a process, physically resident in memory, and free to be used by the process within the restrictions placed by the A and L fields, it is said to be in the active state. The active state is the normal state for a segment that is being used by a process.

6. I/O Ongoing State - When I/O operations are being performed upon the contents of an active segment, the segment is put into the I/O ongoing state. The physical memory being occupied by the segment may not be deallocated and reassigned to another segment. The segment should not be unloaded until all I/O operations terminate.

   A segment should be in I/O ongoing state until all I/O operations being performed upon the contents of the segment have been completed. At this point, the segment may be returned to the active state.

7. Unload Pending State - If it is determined that a segment is to be unloaded and the segment is in the I/O ongoing state, the segment must be placed in an unload pending state which indicates that it is to be unloaded when all I/O operations terminate. When the last I/O operation completes, the segment may be placed in an unloading state and may then be unloaded. If the decision to unload the segment is changed while the segment is in an unload pending state, the state should be changed back to either I/O ongoing, if there are still outstanding I/O operations, or active, if all I/O operations have completed.

b. Reference History Bits - Bits 34 and 35 of the STE (bits 2 and 3 of the SSTE) are called the reference history (R) bits. The H field is used to manage replacement algorithms. At given intervals, the state of the R and D bits in the HSTE are recorded in the H field and are reset in the HSTE.

The state of the R bit is retained only between intervals. For example, if the R bit is reset at the time it is examined, the H field will indicate that the last state of the R bit was reset (0). In contrast to this, once D has been set in the HSTE, that fact is retained in the H field until the segment is either unloaded or a copy of the modified state of the segment is made.

The H field is composed of 2 bits. The most significant bit of the H field will always be set equal to the value of R at the time the HSTE was last scanned and reset.

The least significant bit of the H field will be an OR of its previous contents and the setting of D in the HSTE the last time D was scanned. This results in four possible values for the H field:

1. segment unreferenced in last interval, unmodified (H=00)
2. segment referenced in last interval, unmodified (H=10)
3. segment referenced in last interval, modified (H=11)
4. segment unreferenced in last interval, modified (at some previous time) (H=01)

c. Reserved Field - Bits 36 and 37 of the STE (bits 4 and 5 of the SSTE) are reserved. These bits must be set to 0.

d. Disk Address - Bits 38 through 63 of the STE (bits 6 through 31 of the SSTE) contain the Disk Address (DA) field. This field contains two subfields: the Paging Unit Number (PUN) which is contained in bits 38 through 43 of the STE (bits 6 through 11 of the SSTE) and the Relative Sector Number (RSN) which is contained in bits 44 through 63 of the STE (bits 12 through 31 of the SSTE).

A paging unit is a randomly accessible device which may be read from or written to. This unit is used to load and unload segments. The PUN is used as an index into a Page Device Table (PDT) which is used to translate the PUN into a physical device. The PDT entry contains a physical device address and a device starting sector. The RSN in the SSTE is added to the starting sector specified in the PDT entry to compute the actual sector number at which segment can be found.

There may be up to 32 paging units used in a system at any given time. The PDT allows independence of the logical paging unit from the physical paging unit. For example, a given physical device could be divided into multiple paging units or several physical devices could be combined to be a single paging unit.

11.4 MEMORY ADDRESS TRANSLATOR FAULTS

Previous sections of this manual have stated that certain conditions result in MAT faults. A fault is an indication that some exception condition has occurred and that system software should take some action in response. Some faults (such as access violation) are indicative of error, while other faults (such as presence fault) may be used for management of the software system.

11.4.1 Conditions that Cause MAT Faults

The conditions that cause MAT faults to occur are described in the following sections.

11.4.1.1 PST or SST Size Exceeded Fault

The LPSTD or LDPS instruction defines the MVPA (as in Section 11.3.2.2). If an address is generated that is greater than the MVPA, a PST size exceeded fault occurs.

The LSSTD instruction defines a size for the SST. If an STE in the PST specifies an SST offset greater than the size of the SST, an SST size exceeded fault occurs.

If the MVPA is exceeded for the PST, then no reference to memory is made. If the fault is caused by exceeding the valid size of the SST, then only the Process Segment Table will have been referenced.

11.4.1.2 Nonpresence Fault

The nonpresence fault occurs when an STE which has its presence bit reset (0) is referenced. The program address that caused the fault is returned to systems software. The R bit of the referenced STE is set; but the contents of the segment and the D bit in the STE are not modified as a result of a reference to a nonpresent segment.

If the nonpresent segment can be loaded, the instruction that caused the fault may be reexecuted after the segment is loaded. For certain instructions, software intervention may be required to allow correct reexecution. Section 11.4.4 contains a detailed description of how to recover from a nonpresence fault.

11.4.1.3 Access Level Fault

An access level fault occurs when the access level specified by bits 10 and 11 of the PSW is less than the access level specified in an STE that is referenced. The R bit of the referenced STE is set: the contents of the segment and the D bit in the STE are not modified as a result of a reference for the segment which causes an access level fault.

If system software can correct the fault, the faulting instruction may be reexecuted with certain restrictions. See Section 11.4.4 for details.

11.4.1.4 Access Mode Faults

There are three access mode faults: read access fault, write access fault, and execute access fault. Each fault occurs when a mode of access is attempted for a segment that does not allow that mode of access.

The R bit of the referenced STE is set; but the contents of the segment and the D bit in the STE are not modified as the result of an attempted access resulting in the access mode fault.

If system software can correct the fault, the instruction may be reexecuted with certain restrictions. See Section 11.4.4 for details.

11.4.1.5 Segment Limit Fault

A segment limit fault occurs when the value contained in bits 8:12 of a program address is greater than the value specified in the SLF field of the HSTE. The R bit of the referenced STE is set; but the contents of the segment and the D bit in the STE are not modified as the result of an attempted access resulting in a segment limit fault.

If the system software can correct the fault, then the instruction that caused the fault may be reexecuted with certain restrictions. See Section 11.4.4 for details.

11.4.2 Fault Precedence

While some faults may physically be checked for in parallel by the hardware, there is a logical priority in which faults are checked:

1. Segment table size exceeded
2. Nonpresent segment
3. Segment limit violation
4. Access level violation
5. Access mode violation

Detection of any of the listed MAT faults causes the user instruction to be aborted immediately. The reason for the abort is reported to system software as detailed in Section 11.4.3. Only one MAT fault can occur for a single memory operation.

11.4.3 MAT Fault Handling Routine

When a MAT fault occurs, the MAT fault handling routine pointed to by the MAT fault handler new PSW is entered. The MAT fault interrupt new PSW is contained in physical location X'000090'.

The PSW in effect at the time the fault occurs is placed in registers 14 and 15 of the set selected by the MAT fault handler new PSW. The location counter of the old PSW (register 15) contains the address of the instruction that caused the fault.

Register 13 of the selected set is loaded with a value to indicate the reason for the fault. The possible values are:

VALUE	MEANING
0	Reserved code
1	Access mode fault - execute protected
2	Access mode fault - write protected
3	Access mode fault - read protected
4	Access level fault
5	Segment limit fault
6	Nonpresent segment fault
7	SST size exceeded
8	PST size exceeded

Register 12 of the selected set is loaded with the program address that caused the fault.

If the fault occurred during execution of the LM instruction, the calculated address of the start of the data block is placed in register 11 of the selected set.

11.4.4 Reexecution of Faulting Instructions

In general, an instruction that caused a correctable MAT fault can simply be reexecuted after the fault is corrected.

The Load Multiple (LM) instruction in some cases cannot simply be reexecuted, but must be simulated. When a Load Multiple instruction faults, register 11 of the set specified by the MAT interrupt new PSW is loaded with the program address calculated by the hardware as the effective second operand address of the instruction. If that address is the same as the program address which caused the fault (contained in register 12), then the instruction may be reexecuted once the fault has been corrected; no registers were modified by the LM instruction.

If the addresses in register 11 and register 12 are not equal, at least one register was modified by the LM instruction. Once the fault has been corrected, system software should build and execute an instruction sequence to load the required registers, using the calculated program address in register 11.

If the addresses are not equal, then the difference in the addresses, D, should be computed. The last register modified, M=(D/4)-1+R1, should be calculated. If M is less than the X2 field in an RX1 or RX2, or is less than both the FX2 and SX2 fields in an RX3, the instruction may be reexecuted. If this is not the case, then system software must build an instruction sequence to load the remaining registers from the appropriate memory locations. The location portion of the old PSW should then be incremented by the length of the faulted instruction. At this point, normal execution can be resumed by loading the old PSW.

11.4.5 Effect of System Initialization on the MAT

When the Initialize switch (INIT) on the display panel is depressed, or the processor is powered up, all segmentation, relocation, protection, and MAT interrupts are disabled regardless of the state of bit 21 in the current PSW. The contents of the Shared Segment Table and Process Segment Table descriptor registers must be restored by software after power fail.

The MAT remains disabled until an LPSTD instruction is issued. At this time, the MAT is enabled or remains disabled, according to the state of bit 21 of the current PSW.

11.5 MEMORY MANAGEMENT INSTRUCTIONS

Instructions are provided to control the MAT. These instructions are:

LPSTD Load Process Segment Table Descriptor

LSSTD Load Shared Segment Table Descriptor

11.5.1 Load Process Segment Table Descriptor (LPSTD)

Assembler Notation		Op-Code	Format
LPSTD	D2(X2)	DF1	RX3
LPSTD	A2(FX2,SX2)	DF1	RX3

Operation

The operand specifies the address of the fullword Process Segment Table Descriptor. This descriptor is loaded and its contents define the Process Segment Table to be used in program to physical address translation when bit 21 of the PSW is set.

Condition Code

Unchanged

Programming Notes

This instruction is a privileged instruction.

The address specified by the operand must be on a fullword boundary.

A Process Segment Table Descriptor may be loaded while PSW bit 21 is set or zero.

LPSTD is an extended PSF mnemonic.

11.5.2 Load Shared Segment Table Descriptor (LSSTD)

Assembler Notation		Op-Code	Format
LSSTD	D2(X2)	DF2	RX1,RX2
LSSTD	A2(FX2,SX2)	DF2	RX3

Operation

The operand specifies the address of the fullword Shared Segment Table Descriptor. This descriptor is loaded and its contents define the Shared Segment Table to be used in program to physical address translation when bit 21 of the PSW is set.

Condition Code

Unchanged

Programming Notes

This instruction is a privileged instruction.

The address specified by the operand must be on a fullword boundary.

A Process Segment Table Descriptor may be loaded while PSW bit 21 is set or zero.

LSSTD is an extended PSF mnemonic.

Following an LSSTD instruction, the Process Segment Table Descriptor must be loaded, using the LPSTD or LDPS instruction, before attempting MAT translation with the newly defined shared segment table.

APPENDIX A
CP-CODE MAP

635-1

MSD →

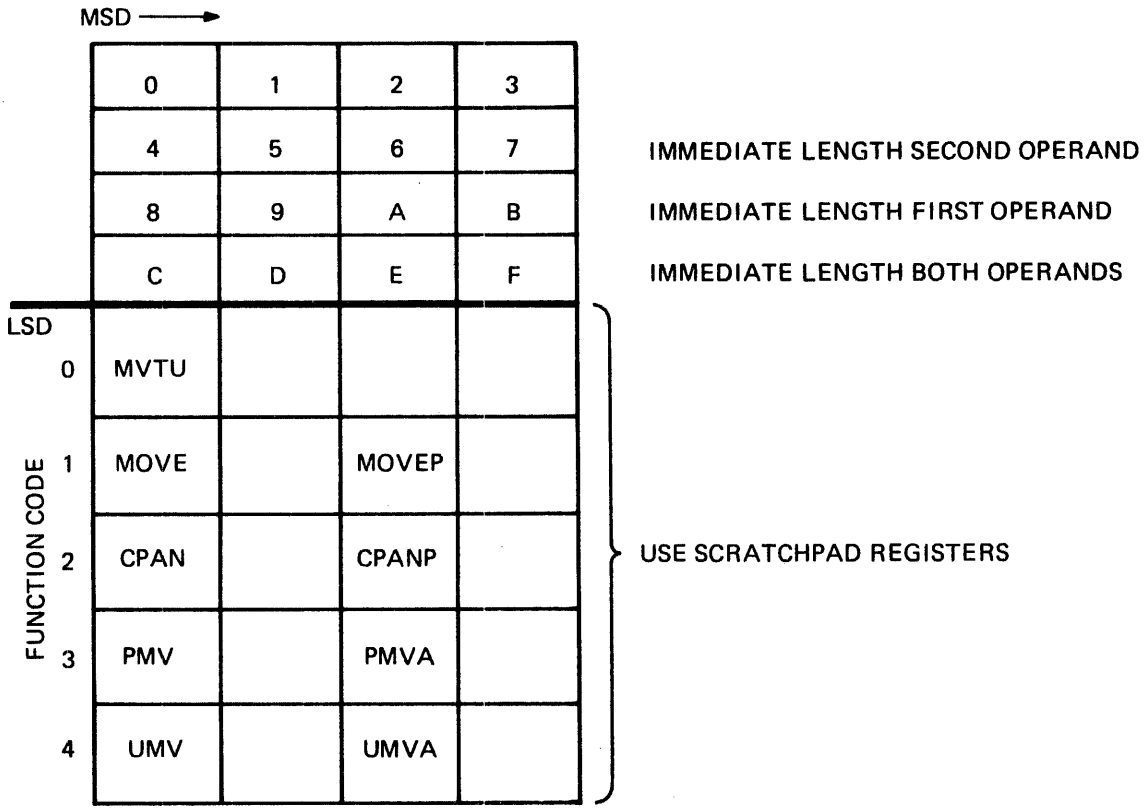
LSD	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0		SRLS	BTBS		STH[4]	ST[3]	STE[3,1]	STD[3,1]		SRHLS			BXH[4]	STM[3]	TS[4]	
1	BALR[4]	SLLS	BTFS		BAL[4]	AM[3]	AHM[4]	STME[3,1]		SLHLS			BXLE[4]	LM[3]	SVC[3]	
2	BTCR[4]	CHVR	BFBS	PBR[2]	BTC[4]		PB[2,4]	LME[3,1]	STDE[3,1]	STBR			LPSW[3,*]	STB	SINT[*]	
3	BFCR[4]	LPER[1]	BFFS	LPDR[1]	BFC[4]		LRA[3]	LHL[4]		LBR			THI	LB	SCP[3,*]	TI
4	NR		LIS	EXHR	NH[4]	N[3]	ATL[3]	TBT	LED[3,1]	EXBR	LEDR[1]		NHI	CLB		NI
5	CLR	LGER[1]	LCS		CLH[4]	CL[3]	ABL[3]	SBT		EPSR[*]	LEGR[1]		CLHI	AL[*]		CLI
6	OR	LGDR[1]	AIS		OH[4]	O[3]	RTL[3]	RBT			LDGR[1]		OHI		LA	OI
7	XR	LCER[1]	SIS	LCDR[1]	XH[4]	X[3]	RBL[3]	CBT	LDE[3,1]		LDER[1]		XHI		TLATE[3]	XI
8	LR	LPSWR[*]	LER[1]	LDR[1]	LH[4]	L[3]	LE[3,1]	LD[3,1]	BRK[*]	WHR[*]			LHI	WH[4,*]		LI
9	CR		CER[1]	CDR[1]	CH[4]	C[3]	CE[3,1]	CD[3,1]		RHR[*]			CHI	RH[4,*]		CI
A	AR		AER[1]	ADR[1]	AH[4]	A[3]	AE[3,1]	AD[3,1]		WDR[*]			AHI	WD[*]	RRL	AI
B	SR		SER[1]	SDR[1]	SH[4]	S[3]	SE[3,1]	SD[3,1]		RDR[*]			SHI	RD[*]	RLL	SI
C	MHR	MR	MER[1]	MDR[1]	MH[4]	M[3]	ME[3,1]	MD[3,1]	RXRX[5]				SRHL		SRL	
D	DHR	DR	DER[1]	DDR[1]	DH[4]	D[3]	DE[3,1]	DD[3,1]		SSR[*]			SLHL	SS[*]	SLL	
E			FXR[1]	FXDR[1]		CRC12[4]	STBP[5]	STMD[3,1]		OCR[*]			SRHA	OC[*]	SRA	
F			FLR[1]	FLDR[1]		CRC16[4]	LPB[5]	LMD[3,1]					SLHA	PSF[*]	SLA	

1. OPTIONAL FLOATING-POINT INSTRUCTION
2. OPTIONAL HIGH-SPEED DATA HANDLING INSTRUCTION
3. SECOND OPERAND ADDRESS MUST BE FULLWORD ALIGNED.
4. SECOND OPERAND ADDRESS MUST BE HALFWORD ALIGNED.
5. USE SCRATCHPAD REGISTERS.

* PRIVILEGED INSTRUCTION

636-1

RXRX SUB FUNCTIONS



PRIVILEGED SYSTEM FUNCTIONS (PSF)

OP-CODE	MNEMONIC	MEANING
DF0	REL	READ ERROR LOGGER
DF1	LPSTD	LOAD PROCESS SEGMENT TABLE DESCRIPTOR
DF2	LSSTD	LOAD SHARED SEGMENT TABLE DESCRIPTOR
DF3	STPS	SAVE PROCESS STATE
DF4	LDPS	LOAD PROCESS STATE
DF5	ISSV	SAVE INTERRUPTIBLE STATE
DF6	ISRST	RESTORE INTERRUPTIBLE STATE
DF7	XSTB	STORE BYTE WITHOUT ECC
DF8	RMVF	RESET MEMORY VOLTAGE FAILURE

APPENDIX B
INSTRUCTION SUMMARY - ALPHABETICAL BY MNEMONIC

MNEMONIC	OP-CODE	INSTRUCTION
A	5A	Add
ABL	65	Add to Bottom of List
AD	7A	Add DPFP
ADR	3A	Add DPFP Register
AE	6A	Add SPFP
AER	2A	Add SPFP Register
AH	4A	Add Halfword
AHI	CA	Add Halfword Immediate
AHM	61	Add Halfword to Memory
AI	FA	Add Immediate
AIS	26	Add Immediate Short
AL	D5	Autoload
AM	51	Add to Memory
AR	0A	Add Register
ATL	64	Add to Top of List
B	430	Branch Unconditional
BAL	41	Branch and Link
BALR	01	Branch and Link Register
BC	428	Branch on Carry
BCR	028	Branch on Carry Register
BCS	208	Branch on Carry Short (Backward)
BCS	218	Branch on Carry Short (Forward)
BE	433	Branch on Equal
BER	033	Branch on Equal Register
BES	223	Branch on Equal Short (Backward)
BES	233	Branch on Equal Short (Forward)
BFBS	22	Branch on False Condition Backward Short
BFC	43	Branch on False Condition
BFCR	03	Branch on False Condition Register
BFFS	23	Branch on False Condition Forward Short
BL	428	Branch on Low
BLR	028	Branch on Low Register
BLS	208	Branch on Low Short (Backward)
BLS	218	Branch on Low Short (Forward)
BM	421	Branch on Minus
BMR	021	Branch on Minus Register
BMS	201	Branch on Minus Short (Backward)
BMS	211	Branch on Minus Short (Forward)
BNC	438	Branch on No Carry

MNEMONIC	OP-CODE	INSTRUCTION
BNCR	038	Branch on No Carry Register
BNCS	228	Branch on No Carry Short (Backward)
BNCS	238	Branch on No Carry Short (Forward)
BNE	423	Branch on Not Equal
BNER	023	Branch on Not Equal Register
BNES	203	Branch on Not Equal Short (Backward)
BNES	213	Branch on Not Equal Short (Forward)
BNL	438	Branch on Not Low
BNLR	038	Branch on Not Low Register
BNLS	228	Branch on Not Low Short (Backward)
BNLS	238	Branch on Not Low Short (Forward)
BNM	431	Branch on Not Minus
BNMR	031	Branch on Not Minus Register
BNMS	221	Branch on Not Minus Short (Backward)
BNMS	231	Branch on Not Minus Short (Forward)
BNO	434	Branch on No Overflow
BNOR	034	Branch on No Overflow Register
BNOS	224	Branch on No Overflow Short (Backward)
BNOS	234	Branch on No Overflow Short (Forward)
BNP	432	Branch on Not Plus
BNPR	032	Branch on Not Plus Register
BNPS	222	Branch on Not Plus Short (Backward)
BNPS	232	Branch on Not Plus Short (Forward)
BNZ	423	Branch on Not Zero
BNZR	023	Branch on Not Zero Register
BNZS	203	Branch on Not Zero Short (Backward)
BNZS	213	Branch on Not Zero Short (Forward)
BO	424	Branch on Overflow
BOR	024	Branch on Overflow Register
BOS	204	Branch on Overflow Short (Backward)
BOS	214	Branch on Overflow Short (Forward)
BP	422	Branch on Plus
BPR	022	Branch on Plus Register
BPS	202	Branch on Plus Short (Backward)
BPS	212	Branch on Plus Short (Forward)
BR	030	Branch Unconditional Register
BRK	88	Breakpoint
BS	220	Branch Unconditional Short (Backward)
BS	230	Branch Unconditional Short (Forward)
BTBS	20	Branch on True Condition Backward Short
BTC	42	Branch on True Condition
BTCR	02	Branch on True Condition Register
BTFS	21	Branch on True Condition Forward Short
BXH	C0	Branch on Index High
BXLE	C1	Branch on Index Low or Equal
BZ	433	Branch on Zero
BZR	033	Branch on Zero Register
BZS	223	Branch on Zero Short (Backward)
BZS	233	Branch on Zero Short (Forward)

MNEMONIC	OP-CODE	INSTRUCTION
C	59	Compare
CBT	77	Complement Bit
CD	79	Compare Double Floating-Point
CDR	39	Compare Double Floating-Point Register
CE	69	Compare Floating-Point
CER	29	Compare Floating-Point Register
CH	49	Compare Halfword
CHI	C9	Compare Halfword Immediate
CHVR	12	Convert Halfword Value Register
CI	F9	Compare Immediate
CL	55	Compare Logical
CLB	D4	Compare Logical Byte
CLH	45	Compare Logical Halfword
CLHI	C5	Compare Logical Halfword Immediate
CLI	F5	Compare Logical Immediate
CLR	05	Compare Logical Register
CPAN	8C/02	Compare Alphanumeric
CPANP	8C/22	Compare Alphanumeric and Pad
CR	09	Compare Register
CRC12	5E	Cyclic Redundancy Check Modulo 12
CRC16	5F	Cyclic Redundancy Check Modulo 16
D	5D	Divide
DD	7D	Divide Double-Precision Floating-Point
DDR	3D	Divide Double Floating-Point Register
DE	6D	Divide Floating-Point
DER	2D	Divide Floating-Point Register
DH	4D	Divide Halfword
DHR	0D	Divide Halfword Register
DR	1D	Divide Register
EPSR	95	Exchange Program Status Register
EXBR	94	Exchange Byte Register
EXHR	34	Exchange Halfword Register
FLR	2F	Float Register
FLDR	3F	Float Register Double Precision
FXDR	3E	Fix Register Double-Precision Floating Point
FXR	2E	Fix Register
ISRST	DF6	Interruptible State Restore
ISSV	DF5	Interruptible State Save
L	58	Load
LA	E6	Load Address
LB	D3	Load Byte
LBR	93	Load Byte Register
LCDR	37	Load Complement Double Floating Register

MNEMONIC	OP-CODE	INSTRUCTION
LCER	17	Load Complement Floating-Point Register
LCS	25	Load Complement Short
LD	78	Load Double-Precision Floating-Point
LDE	87	Load Double Floating-Point From Single
LDER	A7	Load Double From Single Register
LDGR	A6	Load Double From General Register
LDPS	DF4	Load Process State
LDR	38	Load Double-Precision Register
LE	68	Load Floating-Point
LED	84	Load Floating From Double Precision
LEDR	A4	Load Floating From Double Register
LEGR	A5	Load Floating From General Register
LER	28	Load Floating-Point Register
LH	48	Load Halfword
LHI	C8	Load Halfword Immediate
LHL	73	Load Halfword Logical
LI	F8	Load Immediate
LIS	24	Load Immediate Short
LM	D1	Load Multiple
LMD	7F	Load Multiple Double-Precision Floating Point
LME	72	Load Multiple Floating-Point
LPB	6F	Load Packed Binary
LPDR	33	Load Positive Double Floating Register
LPER	13	Load Positive Floating Register
LPSTD	DF1	Load Process Segment Table Description
LPSW	C2	Load Program Status Word
LPSWR	18	Load Program Status Word Register
LR	08	Load Register
LRA	63	Load Real Address
LSSTD	DF2	Load Shared Segment Table Descriptor
M	5C	Multiply
MD	7C	Multiply Double Floating-Point
MDR	3C	Multiply Double Floating Register
ME	6C	Multiply Floating-Point
MER	2C	Multiply Floating-Point Register
MR	4C	Multiply Halfword
MHR	0C	Multiply Halfword Register
MOVE	8C/01	Move
MOVEP	8C/21	Move and Pad
MR	1C	Multiply Register
N	54	AND
NH	44	AND Halfword
NHI	C4	AND Halfword Immediate
NI	F4	AND Immediate
NOP	420	No Operation
NOPR	020	No Operation Register
NR	04	AND Register

MNEMONIC	OP-CODE	INSTRUCTION
O	56	OR
OC	DE	Output Command
OCR	9E	Cutput Command Register
OH	46	OR Halfword
OHI	C6	OR Halfword Immediate
OI	F6	OR Immediate
OR	06	CR Register
PB	62	Prccess Byte
PBR	32	Prccess Byte Register
PMV	8C/03	Pack and Move
PMVA	8C/23	Pack and Move Absolute
RBL	67	Remove from Bottom of List
RBT	76	Reset Bit
RD	DB	Read Data
RDR	9B	Read Data Register
REL	DF0	Read Error Logger
RH	D9	Read Halfword
RHR	99	Read Halfword Register
RLL	EB	Rotate Left Logical
RMVF	DF8	Reset Memory Voltage Fault
RRL	EA	Rotate Right Logical
RTL	66	Remove from Top of List
S	5B	Subtract
SBT	7S	Set Bit
SCP	E3	Simulate Channel Program
SD	7B	Subtract Double-Precision Floating Pcint
SDR	3B	Subtract Register Double-Precision Floating-Point
SE	6B	Subtract Floating-Point
SER	2B	Subtract Floating-Point Register
SH	4B	Subtract Halfword
SHI	CB	Subtract Halfword Immediate
SI	FB	Subtract Immediate
SINT	E2	Simulate Interrupt
SIS	27	Subtract Immediate Short
SLA	EF	Shift Left Arithmetic
SLHA	CF	Shift Left Halfword Arithmetic
SLHL	CD	Shift Left Halfword Logical
SLHLS	91	Shift Left Halfword Logical Short
SLL	ED	Shift Left Logical
SLLS	11	Shift Left Logical Short

MNEMONIC	OP-CODE	INSTRUCTION
SR	0B	Subtract Register
SRA	EE	Shift Right Arithmetic
SRHA	CE	Shift Right Halfword Arithmetic
SRHL	CC	Shift Right Halfword Logical
SRHLS	90	Shift Right Halfword Logical Short
SRL	EC	Shift Right Logical
SRLS	10	Shift Right Logical Short
SS	DD	Sense Status
SSR	9D	Sense Status Register
ST	50	Store
STB	D2	Store Byte
STBP	6E	Store Binary as Packed
STBR	92	Store Byte Register
STD	70	Store Double-Precision Floating-Point
STDE	82	Store Double-Precision in Single-Precision
STE	60	Store Floating-Point
STH	40	Store Halfword
STM	D0	Store Multiple
STMD	7E	Store Multiple Double-Precision Floating-Point
STME	71	Store Multiple Floating-Point
STPS	DF3	Save Process State
SVC	E1	Supervisor Call
TBT	74	Test Bit
THI	C3	Test Halfword Immediate
TI	F3	Test Immediate
TLATE	E7	Translate
TS	E0	Test and Set
UMV	8C/04	Unpack and Move
UMVA	8C/24	Unpack and Move Absolute
WD	DA	Write Data
WDR	9A	Write Data Register
WH	D8	Write Halfword
WHR	98	Write Halfword Register
X	57	Exclusive OR
XH	47	Exclusive OR Halfword
XHI	C7	Exclusive OR Halfword Immediate
XI	F7	Exclusive OR Immediate
XR	07	Exclusive OR Register
XSTB	DF7	Store Byte, No ECC

APPENDIX C
INSTRUCTION SUMMARY - NUMERICAL

OP-CODE	MNEMONIC	INSTRUCTION
01*	BALR	Branch and Link Register
02*	BTCR	Branch on True Condition Register
03*	BFCR	Branch on False Condition Register
04	NR	AND Register
05	CLR	Compare Logical Register
06	OR	OR Register
07	XR	Exclusive OR Register
08	LR	Load Register
09	CR	Compare Register
0A	AR	Add Register
0B	SR	Subtract Register
0C*	MHR	Multiply Halfword Register
0D*	DHR	Divide Halfword Register
10	SRLS	Shift Right Logical Short
11	SLLS	Shift Left Logical Short
12	CHVR	Convert to Halfword Value Register
13+	LPER	Load Positive Floating-Point
15+	LGER	Load General Register from Floating
16+	LGDR	Load General from Double Floating
17+	LCER	Load Complement Floating Register
18	LPSWR	Load Program Status Word Register
1C*	MR	Multiply Register
1D*	DR	Divide Register
20*	BTBS	Branch on True Condition Backward Short
21*	BTFS	Branch on True Condition Forward Short
22*	BFBS	Branch on False Condition Backward Short
23*	BFFS	Branch on False Condition Forward Short

*Condition code not changed
+Optional instruction

OP-CODE	MNEMONIC	INSTRUCTION
24	LIS	Load Immediate Short
25	LCS	Lcad Complement Short
26	AIS	Add Immediate Short
27	SIS	Subtract Immediate Short
28+	LER	Load
29+	CER	Ccmpare Floating-Point
2A+	AER	Add Floating-Point Register
2B+	SER	Subtract Floating-Point Register
2C+	MER	Multiply Floating-Point Register
2D+	DER	Divide Floating-Point Register
2E+	FXR	Fix Register
2F+	FLR	Float Register
32*+	PBR	Process Byte Register
33+	LPDR	Lcad Positive Double Register
34*	EXHR	Exchange Halfword Register
37+	LCDR	Lcad Complement Double Register
38+	LDR	Lcad Register Double-Precision Floating-Point
39+	CDR	Ccmpare Register Double-Precision Floating-Point
3A+	ADR	Add Register Double-Precision Floating-Point
3B+	SDR	Subtract Register Double-Precision Floating-Point
3C+	MDR	Multiply Register Double-Precision Floating-Point
3D+	DDR	Divide Register Double-Precision Floating-Point
3E+	FXDR	Fix Register Double-Precision Floating-Point
3F+	FLDR	Float Register Double-Precision Floating-Point
40*	STH	Store Halfword

*Condition code not changed
+Optional instruction

OP-CODE	MNEMONIC	INSTRUCTION
41*	BAL	Branch and Link
42*	BTC	Branch on True Condition
43*	BFC	Branch on False Condition
44	NH	AND Halfword
45	CLH	Compare Logical Halfword
46	OH	OR Halfword
47	XH	Exclusive OR Halfword
48	LH	Load Halfword
49	CH	Compare Halfword
4A	AH	Add Halfword
4B	SH	Subtract Halfword
4C*	MH	Multiply Halfword
4D*	DH	Divide Halfword
50*	ST	Store
51	AM	Add to Memory
54	N	AND
55	CL	Compare Logical
56	O	CR
57	X	Exclusive OR
58	L	Load
59	C	Compare
5A	A	Add
5B	S	Subtract
5C*	M	Multiply
5D*	D	Divide
5E*	CRC12	Cyclic Redundancy Check Modulo 12
5F*	CRC16	Cyclic Redundancy Check Modulo 16
60*+	STE	Store Floating-Point
61	AHM	Add Halfword to Memory
62*+	PB	Process Byte
63	LRA	Load Read Address
64	ATL	Add to Top of List
65	ABL	Add to Bottom of List

*Condition code not changed
+Optional instruction

OP-CODE	MNEMONIC	INSTRUCTION
66	RTL	Remove from Top of List
67	RBL	Remove from Bottom of List
68+	LE	Load Floating-Point
69+	CE	Compare Floating-Point
6A+	AE	Add Floating-Point
6B+	SE	Subtract Floating-Point
6C+	ME	Multiply Floating-Point
6D+	DE	Divide Floating-Point
6E	STBP	Store Binary as Packed
6F	LPB	Load Packed Binary
70*+	STD	Store Double-Precision Floating-Point
71*+	STME	Store Floating-Point Multiple
72*+	LME	Load Floating-Point Multiple
73	LHL	Load Halfword Logical
74	TBT	Test Bit
75	SBT	Set Bit
76	RBT	Reset Bit
77	CBT	Complement Bit
78+	LD	Load Double-Precision Floating-Point
79+	CD	Compare Double-Precision Floating-Point
7A+	AD	Add Double-Precision Floating-Point
7B+	SD	Subtract Double-Precision Floating-Point
7C+	MD	Multiply Double-Precision Floating-Point
7D+	DD	Divide Double-Precision Floating-Point
7E*+	STMD	Store Multiple Double-Precision Floating-Point
7F*+	LMD	Load Multiple Double-Precision Floating Point
82*+	STDE	Store Double Precision in Single
84+	LED	Load Floating from Double Precision
87+	LDE	Load Double from Floating-Point
88*	BRK	Breakpoint

*Condition code not changed
+Optional instruction

OP-CODE	MNEMONIC	INSTRUCTION
8C	(RXRX)	RXRX Class designator
8C/00	MVTU	Move Translated Until
8C/01	MOVE	Move
8C/02	CPAN	Compare Alphanumeric
8C/03	PMV	Pack and Move
8C/04	UMV	Unpack and Move
8C/21	MOVEP	Move and Pad
8C/22	CPANP	Compare Alphanumeric and Pad
8C/23	PMVA	Pack and Move Absolute
8C/24	UMVA	Unpack and Move Absolute
90	SRHLS	Shift Right Halfword Logical Short
91	SLHLS	Shift Left Halfword Logical Short
92*	STBR	Store Byte Register
93*	LBR	Load Byte Register
94*	EXBR	Exchange Byte Register
95	EPSR	Exchange Program Status Word
98	WHR	Write Halfword Register
99	RHR	Read Halfword Register
9A	WDR	Write Data Register
9B	RDR	Read Data Register
9D	SSR	Sense Status Register
9E	OCR	Output Command Register
A4+	LEDR	Load Floating from Double Register
A5+	LEGR	Load Floating from General Register
A6+	LDGR	Load Double from General Register
A7+	LDER	Load Double from Floating Register
C0*	BXH	Branch on Index High
C1*	BXLE	Branch on Index Low or Equal
C2	LPSW	Load Program Status Word
C3	THI	Test Halfword Immediate
C4	NHI	AND Halfword Immediate
C5	CLHI	Compare Logical Halfword Immediate

*Condition code not changed
+Optional instruction

OP-CODE	MNEMONIC	INSTRUCTION
C6	OHI	OR Halfword Immediate
C7	XHI	Exclusive OR Halfword Immediate
C8	LHI	Lcad Halfwocrd Immediate
C9	CHI	Ccmpare Halfword Immediate
CA	AHI	Add Halfwcrd Immediate
CB	SHI	Subtract Halfword Immediate
CC	SRHL	Shift Right Halfword Logical
CD	SLHL	Shift Left Halfword Logical
CE	SRHA	Shift Right Halfword Arithmetic
CF	SLHA	Shift Left Halfword Arithmetic
D0*	STM	Store Multiple
D1*	LM	Lcad Multiple
D2*	STB	Store Byte
D3*	LB	Lcad Byte
D4	CLB	Compare Logical Byte
D5	AL	Autoload
D8	WH	Write Halfword
D9	RH	Read Halfword
DA	WD	Write Data
DB	RD	Read Data
DD	SS	Sense Status
DE	OC	Output Command
DF	(PSF)	PSF Class Designator
DF0	REL	Read Error Logger
DF1*	LPSTD	Lcad Process Segment Table Descriptor
DF2*	LSSTD	Lcad Shared Segment Table Descriptor
DF3*	STPS	Save Process State
DF4	LDPS	Load Process State
DF5*	ISSV	Interruptible State Save
DF6*	ISRST	Interruptible State Restore
DF7*	XSTB	Store Byte, No ECC
DF8*	RMVF	Reset Memory Voltage Fault

*Condition code not changed
+Optional instruction

OP-CODE	MNEMONIC	INSTRUCTION
E0	TS	Test and Set
E1	SVC	Supervisor Call
E2	SINT	Simulate Interrupt
E3	SCP	Simulate Channel Program
E6*	LA	Load Address
E7*	TLATE	Translate
EA	RRL	Rotate Right Logical
EB	RLL	Rotate Left Logical
EC	SRL	Shift Right Logical
ED	SLL	Shift Left Logical
EE	SRA	Shift Right Arithmetic
EF	SLA	Shift Left Arithmetic
F3	TI	Test Immediate
F4	NI	AND Immediate
F5	CLI	Compare Logical Immediate
F6	OI	CR Immediate
F7	XI	Exclusive OR Immediate
F8	LI	Load Immediate
F9	CI	Compare Immediate
FA	AI	Add Immediate
FB	SI	Subtract Immediate

*Condition code not changed
+Optional instruction

APPENDIX D
ARITHMETIC REFERENCES

637-1

TABLE OF POWERS OF TWO

2^n	n	2^{-n}
1	0	1.0
2	1	0.5
4	2	0.25
8	3	0.125
16	4	0.062 5
32	5	0.031 25
64	6	0.015 625
128	7	0.007 812 5
256	8	0.003 906 25
512	9	0.001 953 125
1 024	10	0.000 976 562 5
2 048	11	0.000 488 281 25
4 096	12	0.000 244 140 625
8 192	13	0.000 122 070 312 5
16 384	14	0.000 061 035 156 25
32 768	15	0.000 030 517 578 125
65 536	16	0.000 015 258 789 062 5
131 072	17	0.000 007 629 394 531 25
262 144	18	0.000 003 814 697 265 625
524 288	19	0.000 001 907 348 632 812 5
1 048 576	20	0.000 000 953 674 316 406 25
2 097 152	21	0.000 000 476 837 158 203 125
4 194 304	22	0.000 000 238 418 579 101 562 5
8 388 608	23	0.000 000 119 209 289 550 781 25
16 777 216	24	0.000 000 059 604 644 775 390 625
33 554 432	25	0.000 000 029 802 322 387 695 312 5
67 108 864	26	0.000 000 014 901 161 193 847 656 25
134 217 728	27	0.000 000 007 450 580 596 923 828 125
268 435 456	28	0.000 000 003 725 290 298 461 914 062 5
536 870 912	29	0.000 000 001 862 645 149 230 957 031 25
1 073 741 824	30	0.000 000 000 931 322 574 615 478 515 625
2 147 483 648	31	0.000 000 000 465 661 287 307 739 257 812 5
4 294 967 296	32	0.000 000 000 232 830 643 653 869 628 906 25
8 589 934 592	33	0.000 000 000 116 415 321 826 934 814 453 125
17 179 869 184	34	0.000 000 000 058 207 660 913 467 407 226 562 5
34 359 738 368	35	0.000 000 000 029 103 830 456 733 703 613 281 25
68 719 476 736	36	0.000 000 000 014 551 915 228 366 851 806 640 625
137 438 953 472	37	0.000 000 000 007 275 957 614 183 425 903 320 312 5
274 877 906 944	38	0.000 000 000 003 637 978 807 091 712 951 660 156 25
549 755 813 888	39	0.000 000 000 001 818 989 403 545 856 475 830 078 125
1 099 511 627 776	40	0.000 000 000 000 909 494 701 772 928 237 915 039 062 5

638

TABLE OF POWERS OF SIXTEEN

16^n							n
						1	0
						16	1
						256	2
					4	096	3
					65	536	4
				1	048	576	5
				16	777	216	6
				268	435	456	7
			4	294	967	296	8
			68	719	476	736	9
		1	099	511	627	776	10
		17	592	186	044	416	11
		281	474	976	710	656	12
	4	503	599	627	370	496	13
	72	057	594	037	927	936	14
1	152	921	504	606	846	976	15

DECIMAL VALUES

639

HEXADECIMAL ADDITION AND SUBTRACTION TABLE
EXAMPLES: 5 + A = F; 18 - D = B; A + B = 15

	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10	1
2	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11	2
3	4	5	6	7	8	9	A	B	C	D	E	F	10	11	12	3
4	5	6	7	8	9	A	B	C	D	E	F	10	11	12	13	4
5	6	7	8	9	A	B	C	D	E	F	10	11	12	13	14	5
6	7	8	9	A	B	C	D	E	F	10	11	12	13	14	15	6
7	8	9	A	B	C	D	E	F	10	11	12	13	14	15	16	7
8	9	A	B	C	D	E	F	10	11	12	13	14	15	16	17	8
9	A	B	C	D	E	F	10	11	12	13	14	15	16	17	18	9
A	B	C	D	E	F	10	11	12	13	14	15	16	17	18	19	A
B	C	D	E	F	10	11	12	13	14	15	16	17	18	19	1A	B
C	D	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B	C
D	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	D
E	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	E
F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E	F
	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	

640

HEXADECIMAL MULTIPLICATION AND DIVISION TABLE

EXAMPLES: 5 x 6 = 1E; 75 ÷ D = 9; 58 ÷ 8 = B; 9 x C = 6C

	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
1	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	1
2	2	4	6	8	A	C	E	10	12	14	16	18	1A	1C	1E	2
3	3	6	9	C	F	12	15	18	1B	1E	21	24	27	2A	2D	3
4	4	8	C	10	14	18	1C	20	24	28	2C	30	34	38	3C	4
5	5	A	F	14	19	1E	23	28	2D	32	37	3C	41	46	4B	5
6	6	C	12	18	1E	24	2A	30	36	3C	42	48	4E	54	5A	6
7	7	E	15	1C	23	2A	31	38	3F	46	4D	54	5B	62	69	7
8	8	10	18	20	28	30	38	40	48	50	58	60	68	70	78	8
9	9	12	1B	24	2D	36	3F	48	51	5A	63	6C	75	7E	87	9
A	A	14	1E	28	32	3C	46	50	5A	64	6E	78	82	8C	96	A
B	B	16	21	2C	37	42	4D	58	63	6E	79	84	8F	9A	A5	B
C	C	18	24	30	3C	48	54	60	6C	78	84	90	9C	A8	B4	C
D	D	1A	27	34	41	4E	5B	68	75	82	8F	9C	A9	B6	C3	D
E	E	1C	2A	38	46	54	62	70	7E	8C	9A	A8	B6	C4	D2	E
F	F	1E	2D	3C	4B	5A	69	78	87	96	A5	B4	C3	D2	E1	F
	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	

641

TABLE OF MATHEMATICAL CONSTANTS

CONSTANT	DECIMAL VALUE	HEXADECIMAL VALUE	FLOATING POINT VALUE
			DOUBLE PRECISION
			SINGLE PRECISION
π	3.14159 26535 89793 23846	3.243F 6A88 85A3 08D3	4132 43F6 A888 5A31
$\pi - 1$	0.31830 98861 83790 67154	0.517C C1B7 2722 0A95	4051 7CC1 B727 220B
$\sqrt{\pi}$	1.77245 38509 05516 02730	1.C5BF 891B 4EF6 AA7A	411C 5BF8 91B4 EF6B
Ln π	1.14472 98858 49400 17414	1.250D 048E 7A1B D0BD	4112 B67A E858 4CAA
$\sqrt{3}$	1.73205 08075 68877 29353	1.BB67 AE85 84CA A73B	411B 67AE 8584 CAA7
e	2.71828 18284 59045 23536	2.B7E1 5162 8AED 2A6B	412B 7E15 1628 AED3
e^{-1}	0.36787 94411 71442 32160	0.5E2D 58D8 B3BC DF1B	405E 2D58 D8B3 BCDF
$\sqrt{e}$	1.64872 12707 00128 14683	1.A612 98E1 E069 BC97	411A 6129 8E1E 069C
$\log_{10}e$	0.43429 44819 03251 82765	0.6F2D EC54 9B94 38CB	406F 2DEC 5A9B 9439
$\log_{2}e$	1.44269 50408 88963 40736	1.7154 7652 B82F E177	4117 1547 652B 82FE
γ	0.57721 56649 01532 86061	0.93C4 67E3 7DB0 C7A5	4093 C467 E37D B0C8
Ln γ	-0.54953 93129 81644 82234	-0.8CAE 9BC1 1F5A 5FF4	C08C AE9B C11F 5A60
$\sqrt{2}$	1.41421 35623 73095 04880	1.6A09 E667 F3BC C909	4116 A09E 667F 3BCD
Ln2	0.69314 71805 59945 30942	0.B172 17F7 D1CF 79AC	40B1 7217 F7D1 CF7A
$\log_{10}2$	0.30102 99956 63981 19521	0.4D10 4D42 7DE7 FBCC	404D 104D 427D E7FC
$\sqrt{10}$	3.16227 76601 68379 33199	3.298B 075B 4B6A 5240	4132 98B0 75B4 B6A5
Ln10	2.30258 50929 94045 68402	2.4D76 3776 AAA2 B05C	4124 D763 776A AA2B

FRACTION CONVERSION TABLE

642

Hexadecimal and Decimal Fraction Conversion Table

HALFWORD													
BYTE					BYTE								
BITS 0123		4567			0123				4567				
Hex	Decimal	Hex	Decimal		Hex	Decimal			Hex	Decimal Equivalent			
.0	.0000	.00	.0000	0000	.000	.0000	0000	0000	.0000	.0000	0000	0000	0000
.1	.0625	.01	.0039	0625	.001	.0002	4414	0625	.0001	.0000	1525	8789	0625
.2	.1250	.02	.0078	1250	.002	.0004	8828	1250	.0002	.0000	3051	7578	1250
.3	.1875	.03	.0117	1875	.003	.0007	3242	1875	.0003	.0000	4577	6367	1875
.4	.2500	.04	.0156	2500	.004	.0009	7656	2500	.0004	.0000	6103	5156	2500
.5	.3125	.05	.0195	3125	.005	.0012	2070	3125	.0005	.0000	7629	3945	3125
.6	.3750	.06	.0234	3750	.006	.0014	6484	3750	.0006	.0000	9155	2734	3750
.7	.4375	.07	.0273	4375	.007	.0017	0898	4375	.0007	.0001	0681	1523	4375
.8	.5000	.08	.0312	5000	.008	.0019	5312	5000	.0008	.0001	2207	0312	5000
.9	.5625	.09	.0351	5625	.009	.0021	9726	5625	.0009	.0001	3732	9101	5625
.A	.6250	.0A	.0390	6250	.00A	.0024	4140	6250	.000A	.0001	5258	7890	6250
.B	.6875	.0B	.0429	6875	.00B	.0026	8554	6875	.000B	.0001	6784	6679	6875
.C	.7500	.0C	.0468	7500	.00C	.0029	2968	7500	.000C	.0001	8310	5468	7500
.D	.8125	.0D	.0507	8125	.00D	.0031	7382	8125	.000D	.0001	9836	4257	8125
.E	.8750	.0E	.0546	8750	.00E	.0034	1796	8750	.000E	.0002	1362	3046	8750
.F	.9375	.0F	.0585	9375	.00F	.0036	6210	9375	.000F	.0002	2888	1835	9375
1		2			3				4				

TO CONVERT .ABC HEXADECIMAL TO DECIMAL

Find .A in position 1 .6250

Find .0B in position 2 .0429 6875

Find .00C in position 3 .0029 2968 7500

.ABC Hex is equal to .6708 9843 7500

TO CONVERT .13 DECIMAL TO HEXADECIMAL

1. Find .1250 next lowest to .1300
   subtract -.1250 = .2 Hex
2. Find .0039 0625 next lowest to .0050 0000
   -.0039 0625 = .01
3. Find .0009 7656 2500 .0010 9375 0000
   -.0009 7656 2500 = .004
4. Find .0001 0681 1523 4375 .0001 1718 7500 0000
   -.0001 0681 1523 4375 = .0007
   .0000 1037 5976 5625 = .2147 Hex
5. 13 Decimal is approximately equal to ←

To convert fractions beyond the capacity of table, use techniques below:

HEXADECIMAL FRACTION TO DECIMAL

Convert the hexadecimal fraction to its decimal equivalent using the same technique as for integer numbers. Divide the results by 16^n (n is the number of fraction positions).

Example: $.8A7 = .540771_{10}$

$8A7_{16} = 2215_{10}$

$16^3 = 4096$

$$4096\overline{)2215.000000}\quad .540771$$

DECIMAL FRACTION TO HEXADECIMAL

Collect integer parts of product in the order of calculation.

Example: $.5408_{10} = .8A7_{16}$

.5408
×16
8 ← [8].6528
×16
A ← [10].4448
×16
7 ← [7].1168

INTEGER CONVERSION TABLE

643

Hexadecimal and Decimal Integer Conversion Table

HALFWORD								HALFWORD							
BYTE				BYTE				BYTE				BYTE			
BITS: 0123		4567		0123		4567		0123		4567		0123		4567	
Hex	Decimal	Hex	Decimal	Hex	Decimal	Hex	Decimal	Hex	Decimal	Hex	Decimal	Hex	Decimal	Hex	Decimal
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	268,435,456	1	16,777,216	1	1,048,576	1	65,536	1	4,096	1	256	1	16	1	1
2	536,870,912	2	33,554,432	2	2,097,152	2	131,072	2	8,192	2	512	2	32	2	2
3	805,306,368	3	50,331,648	3	3,145,728	3	196,608	3	12,288	3	768	3	48	3	3
4	1,073,741,824	4	67,108,864	4	4,194,304	4	262,144	4	16,384	4	1,024	4	64	4	4
5	1,342,177,280	5	83,886,080	5	5,242,880	5	327,680	5	20,480	5	1,280	5	80	5	5
6	1,610,612,736	6	100,663,296	6	6,291,456	6	393,216	6	24,576	6	1,536	6	96	6	6
7	1,879,048,192	7	117,440,512	7	7,340,032	7	458,752	7	28,672	7	1,792	7	112	7	7
8	2,147,483,648	8	134,217,728	8	8,388,608	8	524,288	8	32,768	8	2,048	8	128	8	8
9	2,415,919,104	9	150,994,944	9	9,437,184	9	589,824	9	36,864	9	2,304	9	144	9	9
A	2,684,354,560	A	167,772,160	A	10,485,760	A	655,360	A	40,960	A	2,560	A	160	A	10
B	2,952,790,016	B	184,549,376	B	11,534,336	B	720,896	B	45,056	B	2,816	B	176	B	11
C	3,221,225,472	C	201,326,592	C	12,582,912	C	786,432	C	49,152	C	3,072	C	192	C	12
D	3,489,660,928	D	218,103,808	D	13,631,488	D	851,968	D	53,248	D	3,328	D	208	D	13
E	3,758,096,384	E	234,881,024	E	14,680,064	E	917,504	E	57,344	E	3,584	E	224	E	14
F	4,026,531,840	F	251,658,240	F	15,728,640	F	983,040	F	61,440	F	3,840	F	240	F	15
8		7		6		5		4		3		2		1	

TO CONVERT HEXADECIMAL TO DECIMAL

1. Locate the column of decimal numbers corresponding to the left-most digit or letter of the hexadecimal; select from this column and record the number that corresponds to the position of the hexadecimal digit or letter.
2. Repeat step 1 for the next (second from the left) position.
3. Repeat step 1 for the units (third from the left) position.
4. Add the numbers selected from the table to form the decimal number.

EXAMPLE

Conversion of Hexadecimal Value	D34
1. D	3328
2. 3	48
3. 4	4
4. Decimal	3380

To convert integer numbers greater than the capacity of table, use the techniques below:

HEXADECIMAL TO DECIMAL

Successive cumulative multiplication from left to right, adding units position.

Example: $D34_{16} = 3380_{10}$

D = 13
x16
208
3 = + 3
211
x16
3376
4 = +4
3380

TO CONVERT DECIMAL TO HEXADECIMAL

1. (a) Select from the table the highest decimal number that is equal to or less than the number to be converted.
   (b) Record the hexadecimal of the column containing the selected number.
   (c) Subtract the selected decimal from the number to be converted.
2. Using the remainder from step 1(c) repeat all of step 1 to develop the second position of the hexadecimal (and a remainder).
3. Using the remainder from step 2 repeat all of step 1 to develop the units position of the hexadecimal.
4. Combine terms to form the hexadecimal number.

EXAMPLE

Conversion of Decimal Value	3380
1. D	-3328
	52
2. 3	-48
	4
3. 4	-4
4. Hexadecimal	D34

DECIMAL TO HEXADECIMAL

Divide and collect the remainder in reverse order.

Example: $3380_{10} = X_{16}$

16 | 3380 → remainder 4
16 | 211 → 3
16 | 13 → D

$3380_{10} = D34_{16}$

APPENDIX E
I/O REFERENCES

ASCII/HEX CONVERSION TABLE

644-1

BITS				b_6 b_5 b_4	0 0 0	0 0 1	0 1 0	0 1 1	1 0 0	1 0 1	1 1 0	1 1 1
b_3 ↓	b_2 ↓	b_1 ↓	b_0 ↓	MSD / LSD	0	1	2	3	4	5	6	7
0	0	0	0	0	NUL	DLE	SP	0	@	P	`	p
0	0	0	1	1	SOH	DC1	!	1	A	Q	a	q
0	0	1	0	2	STX	DC2	"	2	B	R	b	r
0	0	1	1	3	ETX	DC3	#	3	C	S	c	s
0	1	0	0	4	EOT	DC4	$	4	D	T	d	t
0	1	0	1	5	ENQ	NAK	%	5	E	U	e	u
0	1	1	0	6	ACK	SYN	&	6	F	V	f	v
0	1	1	1	7	BEL	ETB	'	7	G	W	g	w
1	0	0	0	8	BS	CAN	(	8	H	X	h	x
1	0	0	1	9	HT	EM	)	9	I	Y	i	y
1	0	1	0	A	LF	SUB	*	:	J	Z	j	z
1	0	1	1	B	VT	ESC	+	;	K	[	k	{
1	1	0	0	C	FF	FS	,	<	L	\	l	¦
1	1	0	1	D	CR	GS	-	=	M	]	m	}
1	1	1	0	E	SO	RS	.	>	N	^	n	~
1	1	1	1	F	SI	US	/	?	O	_	o	DEL

NUL	Null
SOH	Start of heading
STX	Start of text
ETX	End of text
EOT	End of transmission
ENQ	Enquiry
ACK	Acknowledge
BEL	Audible signal
BS	Backspace
HT	Horizontal tabulation
LF	Line feed
VT	Vertical tabulation
FF	Form feed
CR	Carrier return
SO	Shift out
SI	Shift in
DLE	Data link escape
DC1-4	Device control
NAK	Negative acknowledge
SYN	Synchronous idle
ETB	End of transmission block
CAN	Cancel
EM	End of medium
SUB	Start of special sequence
ESC	Escape
FS	File separator
GS	Group separator
RS	Record separator
US	Unit separator
SP	Space
DEL	Delete/Idle

ASCII/CARD CODE CONVERSION TABLE

645-1

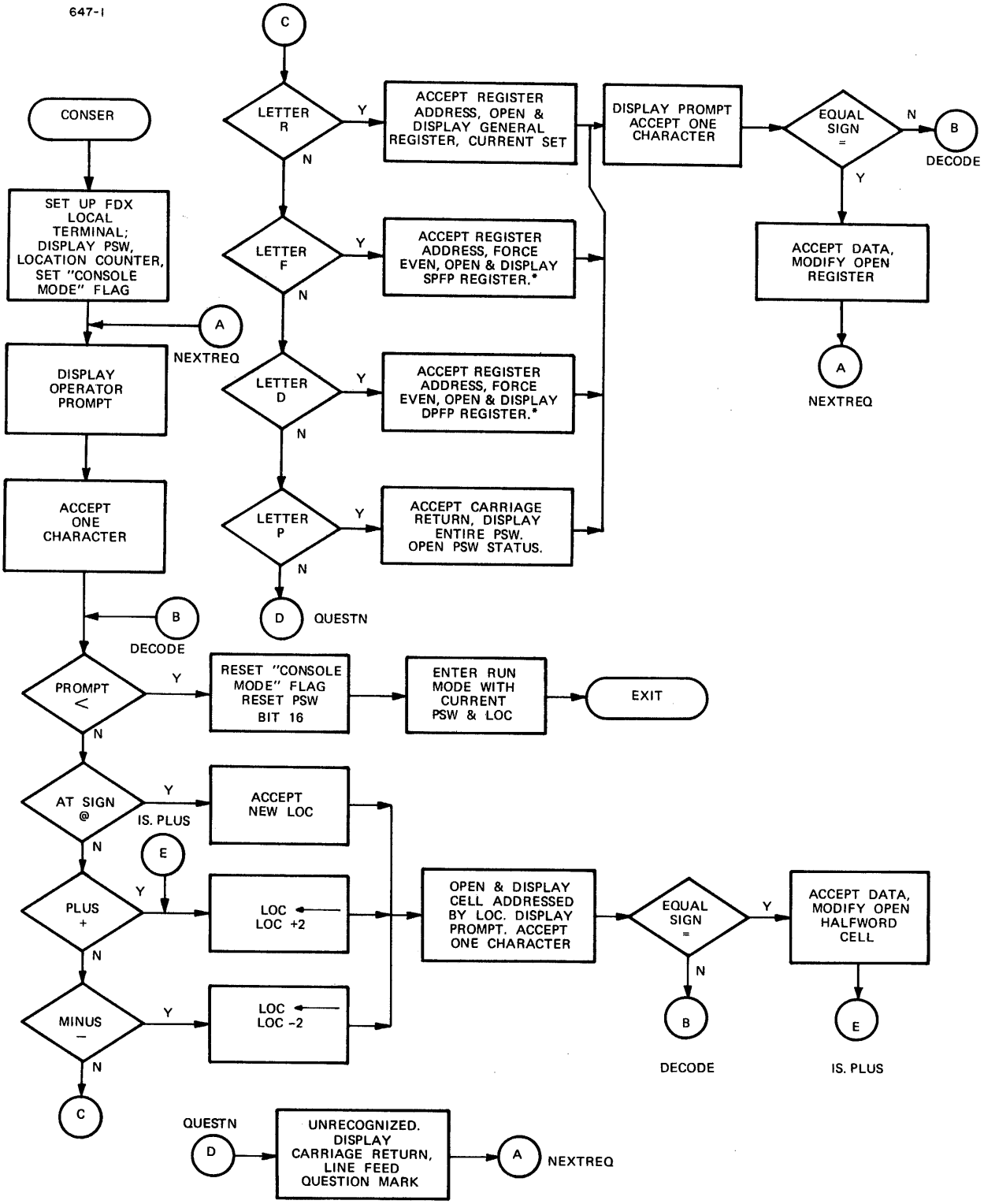
GRAPHIC	7-BIT ASCII CODE	CARD CODE	GRAPHIC	7-BIT ASCII CODE	CARD CODE
SPACE	20	BLANK	@	40	8-4
!	21	11-8-2	A	41	12-1
"	22	8-7	B	42	12-2
#	23	8-3	C	43	12-3
$	24	11-8-3	D	44	12-4
%	25	0-8-4	E	45	12-5
&	26	12	F	46	12-6
'	27	8-5	G	47	12-7
(	28	12-8-5	H	48	12-8
)	29	11-8-5	I	49	12-9
*	2A	11-8-4	J	4A	11-1
+	2B	12-8-6	K	4B	11-2
,	2C	0-8-3	L	4C	11-3
-	2D	11	M	4D	11-4
.	2E	12-8-3	N	4E	11-5
/	2F	0-1	O	4F	11-6
0	30	0	P	50	11-7
1	31	1	Q	51	11-8
2	32	2	R	52	11-9
3	33	3	S	53	0-2
4	34	4	T	54	0-3
5	35	5	U	55	0-4
6	36	6	V	56	0-5
7	37	7	W	57	0-6
8	38	8	X	58	0-7
9	39	9	Y	59	0-8
:	3A	8-2	Z	5A	0-9
;	3B	11-8-6	[	5B	12-8-2
<	3C	12-8-4	\	5C	0-8-2
=	3D	8-6	]	5D	12-8-7
>	3E	0-8-6	↑	5E	11-8-7
?	3F	0-8-7	←	5F	0-8-5

646-I

MSD \ LSD	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0		RESERVED	TTY CAROUSEL 15, 30 CRT ON CLI		CARD READER	LOADER STORAGE UNIT	RESERVED		MDIO						201/301 DATA SET HDX	201/301 DATA SET FDX
1	← COMM MUX →															
2	← 8-LINE INTERRUPT MODULE (ADRS 20 TO 27) →								← SECOND 8-LINE INTERRUPT MODULE (ADRS 28 TO 2F) →							
3		CONTACT CLOSURE MODULE	← I/O BUS SWITCH →												360/370 AUX. INF	360/370 INF
4												DIGITAL MUX				
5																
6			LINE PRINTERS										UNIVERSAL CLOCK VARIABLE 60 Hz			
7		RELAY DRIVER MODULE	↓													801 DIALER
8				CONVERSION EQUIPMENT		556/800 BPI MAG TAPE			AIC			ULI				
9						↓			← AOC →							
A									← DIO →							
B							REMOVABLE CARTRIDGE DISK CONT		← QSA →							
C	MICROBUS ADAPTER	FLOPPY DISK				1600 BPI MAG TAPE	DISK 0	FIXED DISK 0								
D						↓	DISK 1	FIXED DISK 1								
E							DISK 2	FIXED DISK 2								
F	← SELECTOR CHANNELS →						DISK 3	FIXED DISK 3				MSM DISK SYSTEM	DRIVE 0	DRIVE 1	DRIVE 2	DRIVE 3

AIC = ANALOG INPUT CONTROLLER
AOC = ANALOG OUTPUT CONTROLLER
DIO = DIGITAL I/O CONTROLLER
QSA = QUAD SYNCHRONOUS ADAPTER
ULI = UNIVERSAL LOGIC INTERFACE
MDIO = MEMORY DISPATCHED I/O

APPENDIX F
CONSOLE SERVICE ROUTINE FLOWCHART



NOTES:

1. ALL RECEIVED CHARACTERS ECHOED BY PROCESSOR.
2. LOWER-CASE CHARACTERS INTERPRETED AS UPPER-CASE.
3. SPACE CHARACTERS IGNORED.
4. BACKSPACE, UNDERLINE, DELETE CAUSE PREVIOUS NUMERIC CHARACTER TO BE IGNORED.

* IF HARDWARE FLOATING POINT IS NOT AVAILABLE, DISPLAYS ALL "F"s AS CONTENTS.

INDEX

PUBLICATION COMMENT FORM

Please use this postage-paid form to make any comments, suggestions, criticisms, etc. concerning this publication.

From _____ Date _____

Title _____ Publication Title _____

Company _____ Publication Number _____

Address _____

_____

_____

FOLD FOLD

Check the appropriate item.

☐ Error Page No. _____ Drawing No. _____

☐ Addition Page No. _____ Drawing No. _____

☐ Other Page No. _____ Drawing No. _____

Explanation:

FOLD FOLD

CUT ALONG LINE

Fold and Staple

No postage necessary if mailed in U.S.A.

STAPLE

STAPLE

FOLD

FOLD

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 22 OCEANPORT, N.J.

POSTAGE WILL BE PAID BY ADDRESSEE

PERKIN-ELMER

Computer Systems Division
2 Crescent Place
Oceanport, NJ 07757

TECH PUBLICATIONS DEPT. MS 322A

FOLD

FOLD

STAPLE

STAPLE